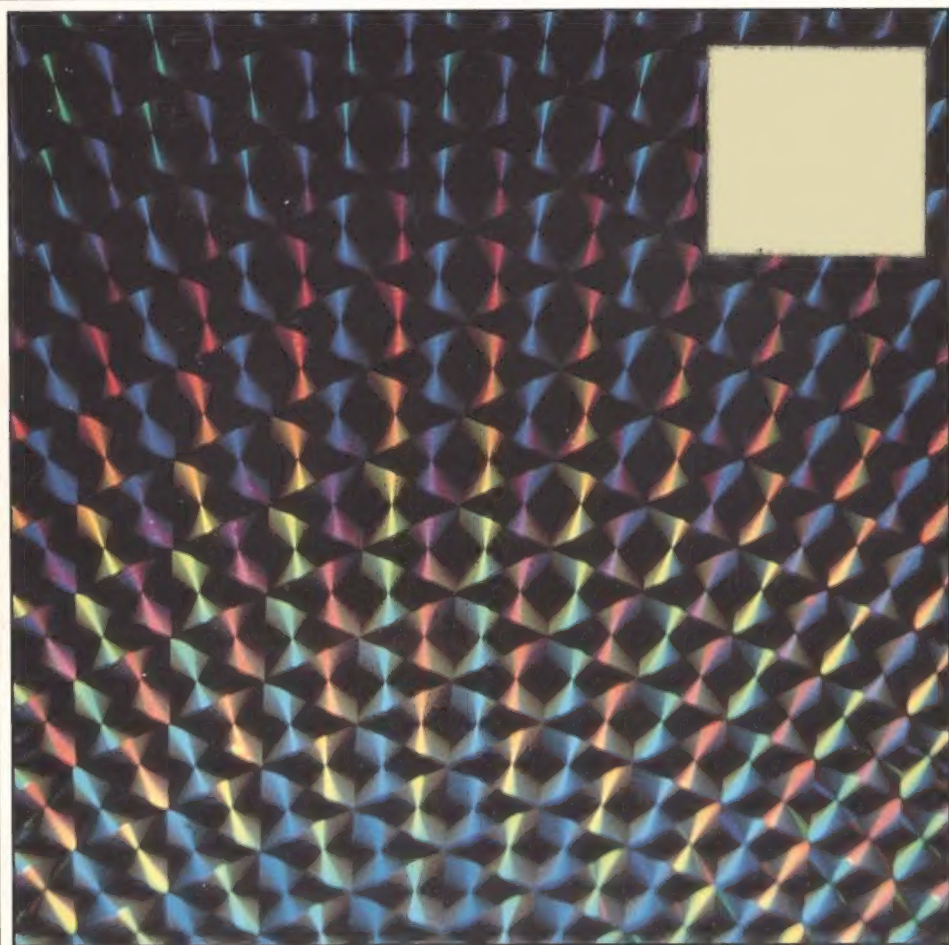




Beyond Basic

6502 Assembly Language Programming for the
British Broadcasting Corporation Microcomputer



Richard Freeman

BEYOND BASIC

6502 Assembly Language on the British Broadcasting
Corporation Microcomputer

Richard Freeman

British Broadcasting Corporation
National Extension College

BEYOND BASIC

6502 Assembly Language on the British Broadcasting
Corporation Microcomputer

Author's acknowledgements

It is a great pleasure to acknowledge the enormous help that Ian Trackman gave me in the preparation of this book. He painstakingly checked through the text and programs, spotting errors and ambiguities; he provided profuse suggestions for additional material and for re-orderings of the material. Whatever its faults, the book is immensely better as a result of Ian Trackman's help.

Errors and faults remain. They are my responsibility and I would welcome letters, from users of the book, pointing out errors and giving suggestions for improvements.

© Richard Freeman 1983

First published 1983

Published by the British Broadcasting Corporation
35 Marylebone High Street
London W1M 4AA

and

National Extension College Trust Ltd
18 Brooklands Avenue
Cambridge CB2 2HN

ISBN 0 563 16592 8 (BBC)

ISBN 0 86082 388 1 (NEC)

Contents

	Page
Introduction	5
How to use this book	8
Unit 1 Number representation	9
Unit 2 Addition and subtraction	28
Unit 3 Jumps, loops and branches	54
Unit 4 Addressing modes	87
Unit 5 Multiplication and division	113
Unit 6 Lists and tables	131
Unit 7 The stack, CALL, USR and masking	171
Unit 8 Operating System calls	204
Unit 9 Tough stuff	225
Unit 10 Round-up	240
Appendix 1 BBC Microcomputer character set	251
Appendix 2 6502 Instruction set	252
Appendix 3 Further reading	256

Introduction

Aims

This book has been written for those people who have a good grasp of BASIC programming and want to move on to 6502 assembly language programming. There are, of course, other books which cover 6502 programming so any new book needs some justification. In the case of this book, it aims to do two things which existing texts do not offer beginners. First, it offers a problem-solving approach to assembly language i.e. it does not work steadily through the instruction set but looks at practical problems which allow each new instruction to be introduced naturally. Second, the book was developed on the BBC Microcomputer. This means that, for British Broadcasting Corporation Micro users, all the programs work as written. They do not leave you to find out obscure features about your micro before you can make the programs run. However, this book will still be of use to owners of other 6502-based micros. Once you know something of your micro's operating system calls, memory map and how to use one of the assemblers available for it, you should be able to enter programs using material from this book.

What is assembly language?

Microprocessors

At the heart of every microcomputer there is a 'microprocessor'. A microprocessor is a silicon chip containing thousands of electronic circuits which together provide the computing power that has made microcomputers hit the headlines. All the other chips in a microcomputer are working in support of the central microprocessor. If you are programming in BASIC, you need not be aware of the microprocessor's existence. But when programming in assembly language, you are much nearer to it. So near, that you need to know which type of microprocessor your computer uses. The BBC Microcomputer has a 6502A microprocessor – hence the sub-title of this book.

Machine code

A microprocessor can only accept instructions in 'machine code'. A program in machine code looks like this (except that the microprocessor is sent the numbers in binary form):

```
A9 19
18
69 18
85 70
60
```

The program would add together the numbers 25 and 27 and store the result in the computer's memory. It's obviously very different from BASIC, particularly because the code appears most unfriendly and

difficult to understand. Yet a microcomputer must provide its microprocessor with instructions in machine code if the microprocessor is to be able to work. On the other hand, because machine code is so unfriendly, it is hard to write machine code. The codes appear to be arbitrary, bearing no apparent relation to what they do (there is a logic to them, but at a level that is not obvious to the human eye). As a result, it is very difficult to program in machine code without making mistakes and those mistakes are very hard to detect. Assembly language has been developed to overcome this problem.

Assembly language

Assembly language is a programming language which allows you to enter instructions which are reasonably meaningful but which the computer can 'assemble' into machine code. The machine code program above resulted from entering the following 'assembly language' program:

```
10 DIM Z%50
20 P% = Z%
30 [
40 LDA #25
50 CLC
60 ADC #27
70 STA &70
80 RTS
90 ]
```

All assemblers work on the same principle and perform the same basic tasks:

- ☐ translate the assembly language instructions into machine code
- ☐ place data in the user's specified memory locations
- ☐ place the machine code program itself in the specified section of memory.

However, although the principles of all assemblers are the same, their detailed workings are not. It is for this reason that this course concentrates on the BBC BASIC assembler.

Once the assembly language program has been entered, you 'assemble' it by typing the command **RUN**. This tells the assembler (which is a computer program) to translate your assembly language program into machine code. At that point you can, if you wish, discard the assembly language program and just keep the machine code program – called the 'object program'.

The BASIC assembler

There are many 6502-based microcomputers. All use the same machine code because the 6502 and 6502A microprocessors only obey 6502 machine code instructions. But a 6502 microcomputer can have a variety of assemblers developed for it and each will have its own method of use. In the case of the BBC Micro, the built-in assembler is a

'BASIC assembler' i.e. you enter the assembly language instructions within a BASIC program, indicating which parts are assembly language by enclosing them in square brackets, thus: [1].

How BASIC is different

BASIC is what is called a 'high level' language and assembly languages are 'low level' languages. In a low level language, you enter instructions at or near to the level at which the microprocessor operates. You talk its language.

In a high level language, what you enter may have very little to do with the language used by the microprocessor. Instead, you work in a language where each instruction (e.g. PRINT or SQR) might require the computer to execute a long series of machine code instructions. In the case of BASIC, the microcomputer executes the instructions by means of a 'BASIC interpreter' – a program in the computer which converts your BASIC program into instructions that the microprocessor can understand; that is, into machine code. An interpreter is not the same as an assembler: an assembler assembles your assembly language program into machine code once and for all and, in the case of some assemblers, shows you the machine code listing. An interpreter has to interpret each BASIC statement every time that you run the program.

Why use assembly language?

First, because it is fast. Because BASIC is interpreted, it is slow – interpreting takes up time.

Second, use it when you are short of memory space. A machine code program often uses less memory space than an equivalent BASIC program to do the same job.

The operating system

You will find a lot of references in this book to the BBC Micro's operating system. This is a program in the computer that controls the machine itself, irrespective of what programming language is currently being used in the computer. For example, the operating system controls display on the screen, and communications with other devices such as printers and cassette recorders. Whatever programming language you are currently using, you need an easy way of using that language to send instructions to the operating system. In BASIC you can use, for example, VDU and *FX. This book shows you how to do the same jobs in assembly language.

And next ...

That's enough general description. The way to learn anything is to do it – so it's now time to get into the book.

How to use this book

This book is a course, not a reference book. In other words, in order to master the ideas in it, you need to work through it. That means trying all the programs, running them with our data and your own until you understand how they work. It also means trying all the 'SAQs':

SAQs

SAQ stands for 'self assessment question'. These questions are designed to help you learn. You can't learn programming by reading about what you should do. You learn by doing things and by asking questions until you get the answers you need. The SAQs help with that process. In this book they are very important since in many cases they contain key ideas. If you skip them, the only way you will find back to these points is through the index.

Assignments

Assignments are for students who are using this book as part of a National Extension College course. The answers to assignments are not included in this book.

Use of colour

Colour is used to highlight important points and new material.



Whenever you see this symbol you should stop reading and writing and get to your keyboard in order to try the activity described.

The tape

Most of the programs in this course are on the cassette tape which accompanies the course. Loading these into your micro will save you a lot of typing and avoid your accidentally introducing typing errors which will prevent your programs from running. Programs are listed on the tape by a code made up from their program number in the course. Thus 'Program 6.3' in the text appears on the tape as 'AS6P3'.

Some programs have made use of Mode 3 to produce a clearer screen layout. If you are using a model A micro, then you will need to change the Mode line in the program to Mode 7 before you can run the program.

Note

The programs in this book will run as written on a BBC Microcomputer. They will also run on an Acorn Electron computer provided you change any 'MODE 7' statements to a mode between 0 and 6. The tape which accompanies this book will load into both the BBC Micro and the Electron.

UNIT 1

Number representation

- 1.1 Number systems and bases
- 1.2 The binary system
- 1.3 Binary numbers in microcomputers
- 1.4 Hexadecimal numbers
- 1.5 Numbers in the BBC Micro
- 1.6 The computer's memory

Assignment A

Objectives of Unit 1

Answers to SAQs

Introduction

Although computers use binary numbers, we only need to use decimal numbers when programming in BASIC. Assembly language programming, on the other hand, requires an understanding of binary and hexadecimal numbers. If you are already familiar with these, including the two's complement method of representing signed numbers, then you may wish to move on to section 1.6 which deals with the memory of the BBC Micro.

1.1 Number systems and bases

There are many methods of representing numbers, some more efficient than others. For example, the Roman method using I, V, X, L, C and M is very inefficient, making even simple arithmetic calculations very tedious. But most number systems involve a 'base', with our everyday number system using a base of 10.

The decimal system

In the decimal system every number is represented by a combination of the ten digits

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

To make a number larger than 9, the digits are positioned to show how many 10's, 100's ($10 * 10$), 1000's ($10 * 10 * 10$) etc. there are in the number. So

$$\begin{aligned}837 &= 800 + 30 + 7 \\&= 8 * 100 + 3 * 10 + 7 \\&= 8 * 10^2 + 3 * 10^1 + 7 * 10^0.\end{aligned}$$

SAQ 1

Write the following in the form $(a * 10^3) + (b * 10^2) + (c * 10^1) + (d * 10^0)$ where a, b, c and d are digits between 0 and 9:

(a) 8352 (b) 5032 (c) 2100

SAQ 2

Write the following as ordinary decimal numbers:

(a) $(4 * 10^3) + (7 * 10^2) + (5 * 10^1) + (8 * 10^0)$

(b) $(8 * 10^3) + (6 * 10^1) + (3 * 10^0)$

(c) $(5 * 10^3)$

This system is the decimal system and is based on the number 10. It uses 10 different digits and is said to be a 'base 10' number system.

Any positive integer can be used as a base. For example, if we write numbers with a base of 5 then we use the five digits 0, 1, 2, 3, 4. Any

number represented in base 5 will be of the form:

$$(a_n * 5^n) + (a_{n-1} * 5^{n-1}) + \dots + (a_1 * 5^1) + (a_0 * 5^0)$$

where a_n, a_{n-1}, \dots, a_1 and a_0 are each one of 0, 1, 2, 3 or 4. The number 48 (decimal) would be 143 (base 5) because

$$143 \text{ (base 5)} = (1 * 5^2) + (4 * 5^1) + (3 * 5^0).$$

Once we start writing numbers in different bases, we need a way of indicating which base we are using. This is done by writing the base as a suffix after the number. Thus

48 (decimal) is written 48_{10} (read as '48 base 10')

and 143 (base 5) is written 143_5 (read as 'one-four-three base 5').

We can therefore write statements such as

$$48_{10} = 143_5.$$

1.2 The binary system

The binary system is numbers to base 2 i.e. every number is written using only the digits 0 and 1. In the binary system, numbers are of the form

$$a_n * 2^n + a_{n-1} * 2^{n-1} + \dots + a_1 * 2^1 + a_0 * 2^0$$

where each of a_n, a_{n-1}, \dots, a_1 , and a_0 is either 0 or 1. The number 27_{10} would be written as

$$11011_2$$

since

$$\begin{aligned} 11011_2 &= (1 * 2^4) + (1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) \\ &= 16 + 8 + 0 + 2 + 1 \\ &= 27. \end{aligned}$$

SAQ 3

What are these binary numbers in decimal?

(a) 1000 (b) 1111 (c) 11111111 (d) 1001010.

Converting decimal to binary involves successive divisions by 2, recording the remainders at each stage. For example, in SAQ3(d) we found that $1001010_2 = 74_{10}$. Let's now reverse the process and start from 74_{10} .

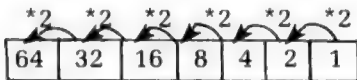
		Remainder
$74 / 2 = 37$	0	
$37 / 2 = 18$	1	(represents $1 * 2^0$)
$18 / 2 = 9$	0	
$9 / 2 = 4$	1	(represents $1 * 2^3$)
$4 / 2 = 2$	0	
$2 / 2 = 1$	0	
$1 / 2 = 0$	1	(represents $1 * 2^6$)

Writing down the remainders from right to left, we find that:

$$74_{10} = 1001010_2.$$

Checking binary numbers

You can check that 1001010 is in fact the correct answer to writing 74_{10} in binary. This you can do by using the place value of the binary place positions. A '1' in a binary number represents 1, 2, 4, 8, 16, ... (decimal) according to where it appears:



Moving from right to left, each digit has double the value of its predecessor. Applying this to 1001010, we find:

$$\begin{array}{r} 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\ 64 \ + \ 8 \ + \ 2 \ = \ 74_{10} \end{array}$$

SAQ 4

Convert the following to binary:

(a) 12_{10} (b) 100_{10} (c) 255_{10} (d) 65535_{10}

Notice that (c) and (d) represent the largest numbers that can be made with 8 binary digits and 16 binary digits respectively. These numbers are important when programming computers and they will be mentioned on many other occasions during this course.

Adding and subtracting

Because only two digits are involved in binary numbers, binary addition and subtraction is very simple.

Addition

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$\begin{aligned}0 + 1 &= 1 \\1 + 1 &= 10\end{aligned}$$

That final 10 means 10_2 (i.e. 2_{10}). It is not the decimal number ten.

Subtraction

$$\begin{aligned}0 - 0 &= 0 \\1 - 0 &= 1 \\0 - 1 &= -1 \\1 - 1 &= 0\end{aligned}$$

Here are some examples of binary addition. First, an addition without a carry:

$$\begin{array}{r}10101 \quad (= 21) \\+ 1010 \quad (= 10) \\ \hline 11111 \quad (= 31)\end{array}$$

Next, an addition that involves carries:

$$\begin{array}{r}11111 \quad (= 31) \\+ 1010 \quad (= 10) \\ \hline 1111 \quad (\text{Carries}) \\101001 \quad (= 41)\end{array}$$

Let's check this result, using the place values of 1, 2, 4, ...

$$\begin{array}{r}1 \ 0 \ 1 \ 0 \ 0 \ 1 \\32 + 8 + 1 = 41_{10}\end{array}$$

Next, subtraction, starting with an example that does not require a borrow:

$$\begin{array}{r}11111 \quad (= 31) \\- 1010 \quad (= 10) \\ \hline 10101 \quad (= 21)\end{array}$$

And finally an example of subtraction with borrows:

$$\begin{array}{r}10101 \quad (= 21) \\- 1111 \quad (= 15) \\ \hline 11 \quad (\text{Borrows}) \\110 \quad (= 6)\end{array}$$

SAQ 5

Give the results in binary number notation of the following calculations:

- (a) $110011 + 1100$ (b) $1000 + 101$ (c) $110011 + 1111$ (d) $11111111 + 1$
(e) $1111 - 110$ (f) $1111 - 101$ (g) $110011 - 1111$ (h) $10000 - 1$

1.3 Binary numbers in microcomputers

Most microcomputers are what is called '8-bit', although 16-bit micros are becoming more widely available. In an 8-bit micro, each unit of memory (usually called a 'memory location' or 'address') has room for eight 0's or 1's. In Figure 1.1, an 8-bit memory location is shown storing the decimal number 103_{10} .

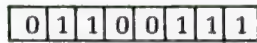


Figure 1.1 An 8-bit memory location storing 103_{10}

If only positive whole numbers were to be stored then the range of numbers which could be stored in one 8-bit location would be 0 to 255. If two memory locations were used to show one number then the range of positive numbers which could be stored would be 0 to 65535. Figure 1.2 shows two 8-bit locations being used to store the decimal number 6836.



Figure 1.2 A 16-bit number (6836_{10}) stored in two 8-bit locations

However many bits we use, the bits are always numbered according to the same system. The 'least significant bit' (often shortened to 'lsb') is called 'bit 0' and the remaining bits are numbered upwards from there. In an 8-bit system, the bit numbers are as shown in Figure 1.3.

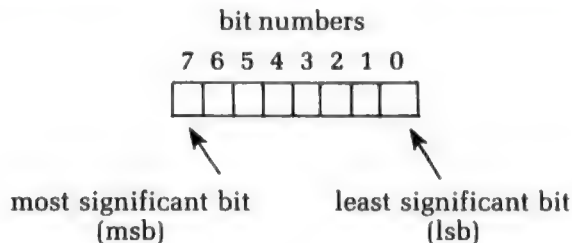


Figure 1.3 Bit numbers in an 8-bit system

We can check that this makes sense by adding the two's complement for 104 to that for -104:

$$\begin{array}{r}
 01101000 \quad (= 104) \\
 \text{add } 10011000 \quad (= -104) \\
 \hline
 10000000 \\
 \uparrow \\
 \text{Zero in 8-bit representation}
 \end{array}$$

Since we fixed all numbers to be 8-bits long, the answer is zero – which is what we would expect from adding +104 to -104. What about the carry? Since it is outside the limits that we have defined for our number system, we just ignore it.

SAQ 6

Which decimal numbers do the following represent? Form their two's complements. Which decimal numbers do the results represent? Verify your answers by adding back the original numbers.

(a) 00100101 (b) 00000001 (c) 00000000 (d) 01111111

If you look at the numbers we have just generated, there is a pattern to their bit 7's:

Positive numbers	Negative numbers
01101000 (+104)	10011000 (-104)
00100101 (+37)	11011011 (-37)
00000001 (+1)	11111111 (-1)
01111111 (+127)	10000001 (-127)

Two's complement numbers
 Positive numbers: bit 7 = 0
 Negative numbers: bit 7 = 1

For this reason, bit 7 is called the 'sign bit'. Because bit 7 is needed to show the sign of the number, there are only 7 bits left (bits 0 to 6) for the magnitude of the number. The largest positive number in an 8-bit system using both positive and negative numbers is:

$$01111111 (= +127)$$

and the smallest number is

$$10000000 (= -128).$$

The range of numbers represented is

$$-128 \text{ to } +127.$$

i.e. 256 numbers. In other words, as you might expect, 8-bits can be used to represent the numbers 0 to 255 in ordinary binary or -128 to +127 in a signed binary system.

What you need to know

How and why the two's complement method of producing signed numbers works requires mathematical proof beyond the scope of this course. All we have done is to show you how to generate two's complement numbers and to give some examples to show that they give correct answers. For the rest of this course all you need to know is:

- ☐ Microcomputers usually use two's complement signed numbers.
- ☐ The msb is the sign bit with 0 = positive and 1 = negative.
- ☐ How to form a two's complement number.
- ☐ That if you correctly perform binary arithmetic on two's complement numbers, correct answers will result.

1.4 Hexadecimal numbers

Microcomputers also use hexadecimal numbers, which are numbers base 16. They are often referred to as 'hex numbers'. To write down numbers to base n we need n digits. That causes no problems for bases up to 10, since we have 10 ready-made digits (0-9). But for hexadecimal we need 16 digits which is 6 more than we already have digits for. We need digits to represent '10', '11', '12', '13', '14' and '15' as digits instead of as base 10 numbers. For these extra digits we use the letters A, B, C, D, E and F. The hexadecimal digits are

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

The positions of the first four digits in a hexadecimal number have the values

4096 256 16 1

Here are some examples:

$$\begin{aligned} 12_{16} &= (1 * 16^1) + (2 * 16^0) \\ &= 16 + 2 \\ &= 18 \end{aligned}$$

$$\begin{aligned} 3DF &= (3 * 16^2) + (13 * 16^1) + (15 * 16^0) \\ &= 768 + 208 + 15 \\ &= 991 \end{aligned}$$

SAQ 7

Write the following hexadecimal numbers as decimal numbers:

(a) E9 (b) 10 (c) 100 (d) FFFF (e) ABCD

The importance of hexadecimal

You can see how tedious binary calculations are. It also difficult to be sure that you have read and transcribed binary numbers correctly. For example, the apparently simple decimal calculation

$$348 + 131 = 479$$

involves writing down 9 digits. In binary, the calculation is

$$101011100 + 10000011 = 111011111$$

and involves 26 digits. Although the computer has to work in binary, it is more convenient if we can communicate with it in a more compact number system. The system we use (when not in decimal) is hexadecimal. It is used because of its very close relationship to binary.

There are 16 hexadecimal digits. If these digits are coded in binary, 4 bits are needed:

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Any 4-bit binary number can be represented by one hexadecimal digit:

1011 can be represented by B

0111 can be represented by 7.

We can represent any 8-bit binary number by two hexadecimal digits e.g.

$$1110\ 1111 = EF \text{ since } 1110 = E \text{ and } 1111 = F.$$

So

$$11101111_2 = EF_{16} = 239_{10}.$$

The simplicity of hexadecimal is that, provided we use binary numbers which are 4 bits long (or multiples of 4 bits long), we can directly exchange 4-bit sections for their hexadecimal equivalents. For this reason, the BBC Micro often allows you to extract information in hexadecimal e.g. the length of a program in bytes or a position in memory.

Hexadecimal numbers are sometimes preceded by the symbol '\$' or followed by the symbol 'H' e.g.

$$\text{\$}25 \text{ and } 25\text{H both mean } 25_{16}$$

So as to avoid any confusion with the string symbol \$, the BBC Micro uses the symbol & for hexadecimal. So

$$\&25 \text{ (BBC Micro)} = \text{\$}25 \text{ (some micros)} = 25_{16}.$$

Throughout this course we shall use '&' to indicate hexadecimal numbers.

SAQ 8

Write the following binary numbers as hexadecimal:

- (a) 01000100 (b) 10000000 (c) 00000001 (d) 11111111
(e) 10101111

1.5 Numbers in the BBC Micro

After all that trouble of converting numbers from one base to another, you will be glad to know that the BBC Micro will convert numbers into certain forms at your request.

To print in decimal

Typing `PRINT 142` produces 142 on the screen. The BBC Micro normally produces decimal in `PRINT` statements.

To convert decimal to hexadecimal

Typing `PRINT "142` produces 8E on the screen. 142_{10} has been converted to hexadecimal. (The symbol '~' is called a tilde. It is the shifted '~' and appears as '~' in Mode 7. In all other Modes it appears correctly on the screen).

To convert hexadecimal to decimal

The BBC Micro will give you output in hexadecimal. If you want to convert this to decimal you use PRINT &... PRINT &8E produces the response 142.

Examples of conversions

>PRINT 142	
142	No conversion. Normal BBC PRINT produces decimal output.
>PRINT &A5	
165	BASIC PRINT normally produces decimal, so &A5 was converted to decimal 165.
>PRINT ~&A5	
A5	The tilde directs the PRINT statement to display in hex. Hence the hex number stays in hex.
>PRINT ~142	
8E	The tilde directs the PRINT statement to display in hex. So decimal 142 is converted to hex.

1.6 The computer's memory

When operating in BASIC, a microcomputer allocates memory space to programs and variables in a manner determined by the language system's designers. You can override the automatic memory allocation and use a system of your own, but you rarely need to do so. In assembly language, on the other hand, you have to make most of the decisions about memory allocation.

The part of the memory we are concerned with is the RAM (random access memory). An outline of this for the BBC Micro is shown in Figure 1.5a (for micros without disc storage). Each memory location is referred to by its position in the memory. The first location is &0000, the next &0001, the next &0002 and so on up to &FFFF. The memory is also notionally sub-divided into 'pages', each &100 (256 decimal) locations long. Blocks of pages are then set aside for specific purposes. Some of these can be seen in Figure 1.5b. Two pages are of special importance in assembly language programming: Zero Page (locations &0000 to &00FF) and Page One (locations &0100 to &01FF).

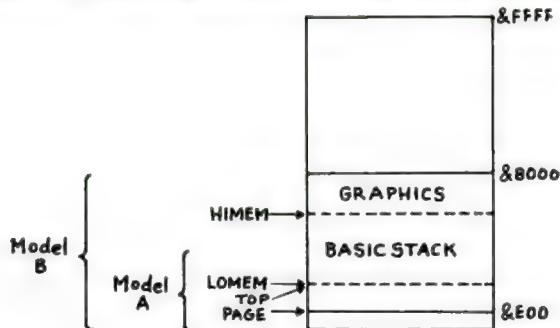


Figure 1.5a The BBC Micro's memory

space for user routines	&E00
user-defined characters	&D00
function keys	&C00
buffers	&B00
buffers	&A00
misc. workspace	&900
language workspace	&800
misc. workspace	&400
OS workspace	&300
6502 stack	&200
Zero Page	&100
	&0

Figure 1.5b Detail of the lower part of the memory

How the memory is used depends on the operating system design. All 6502-based computers use Zero Page in a similar way and Page One in an identical way.

Zero Page

Almost anything done with Zero Page works faster than doing the same thing with another part of the memory. (We will go into the details of why this is later in the course). So the computer designer needs to use a large part of Zero Page for built-in routines of the computer itself. But in the BBC Micro, a part of Zero Page is left free for the user. The free locations are &70 to &8F.

Page One

This is a very special section of memory and is called the 'stack'. It is used by the 6502 microprocessor, largely to keep track of what it is doing. You can use Page One with care but do not attempt to do so until you have studied the Unit on the stack. Any accidental altering of memory values in Page One may cause the computer to lose track of what it is doing and could mean that you lose your program and data.

Other pages

The other pages that you will most immediately need are those in which you can place assembly language programs or data for those programs. First, let's look at how the memory is organised for BASIC programs. In Figure 1.5a you can see four positions in memory: PAGE, TOP, LOMEM and HIMEM. Let's look at each of these:

PAGE

This is usually location &E00 (&1900 in disk-based systems) and is the start address of the BASIC programs that you type in. Your BASIC programs fill memory from &E00 upwards.

TOP

This is the end location of any BASIC program you currently have in memory. So TOP depends on how long your program is. The length of your program is found from **TOP - PAGE**.

LOMEM

LOMEM is the location in memory above which your BASIC program stores its variable values.

HIMEM

This is the highest location in memory in which your BASIC program may store its variables. Unless you specifically alter the position of HIMEM, immediately above it there is an area of memory used for graphics. The higher the resolution of the Mode you are in, the lower the position of HIMEM.

PAGE, LOMEM and HIMEM can all be altered by the programmer.

You should only move PAGE up from &E00 (or &1900 in disc-based systems), not down. If, before entering a BASIC program, you raise PAGE &100 bytes with the instruction

```
PAGE = &F00 (or PAGE = PAGE + &100)
```

then your BASIC program will be entered &100 bytes higher than usual. This means that the locations &E00 to &EFF will not be used by the program and become free for your use.

BASIC variables are usually stored above the program. If you want to create &100 bytes of free memory between the program and LOMEM then type

```
LOMEM = LOMEM + &100
```

That would make a BASIC program store its variables &100 bytes higher than usual, so leaving &100 bytes between TOP and LOMEM for you to use.

You might wish to arrange to store your BASIC variables below the program so that LOMEM would be lower than PAGE. Such an arrangement of memory is not used in this course.

HIMEM may only be moved down from its default position for whichever Mode you are in. You might move HIMEM down with the instruction

```
HIMEM = HIMEM - &100
```

in order to free &100 bytes for your use immediately above HIMEM.

Don't worry if you don't feel confident about moving PAGE, LOMEM and HIMEM at this stage. As the course progresses, you will find 'memory management' easier to follow.

Reading memory values

You may 'read' (i.e. look at) any memory location including Page One (with the exception of certain special Input-Output locations between addresses &FC00 and &FFFF) without doing any harm. (Reading a memory location does not alter its value unless it is a special type of location). Try the following to read any memory location.

To find out what value is currently in a memory location such as location &70, type

```
PRINT ~?&70    (for the contents in hex)
PRINT ?&70      (for the contents in decimal)
```

The use of ? to read the value in a memory location is the BBC Micro's equivalent of the 'PEEK' statement used on most other micros.

[K] Try `PRINT ~?& ...` and `PRINT ?& ...` on some locations on your Micro.

SAQ 9

The result of printing out the contents of a memory location is always a number between 0 and &FF (0 - 255 decimal). Why?

What these numbers represent depends on what the memory locations are being used for: e.g. to represent a number, an ASCII code, a piece of assembly language and so on. You can't tell which of these the number is simply from inspecting the memory in this way.

Altering memory values

To make the memory location &70 contain the decimal number 15, you type

```
?&70 = 15
```

The use of ? to alter the value in a memory location is the BBC Micro's equivalent of the 'POKE' statement on most other micros.

This has put the decimal number 15 into the memory location &70 and the previous contents of &70 have been lost. If you want to put a hexadecimal number – say &42 – into &70 then you type

```
?&70 = &42
```

SAQ 10

(a) Find out what values are currently in locations &70 to &74 by typing `PRINT ~?& ...` and `PRINT ?& ...` for each location.

(b) Now put five numbers of your choice into the locations using `?&70 = ...` etc.

(c) Now repeat (a) to check that the values you have placed in these locations in (b) are really there.

More on memory operations

There are a few more facilities available for placing data in memory from BASIC. These are described in section 6.7. You may like to read that section now although the techniques described are not used until Unit 6.

Assignment A

- 1 What are the following in decimal?
(a) 63_7 (b) 35_6 (c) 101_4 (d) 211_3
- 2 Convert the following decimal numbers to the bases shown.
(a) 57 into base 3 (b) 25 into base 8
- 3 Write the following decimal numbers in binary.
(a) 13 (b) 100 (c) 211
- 4 Evaluate:
(a) $10010 + 101$ (b) $10111 + 101$ (c) $1111 + 1111$
- 5 Evaluate:
(a) $1101101 - 1001$ (b) $1101101 - 11111$ (c) $100000 - 111$
- 6 Put the following decimal numbers into two's complement 8-bit signed format.
(a) 18 (b) -18 (c) 127 (d) -127

Objectives of Unit 1

After studying this Unit you should be able to:

- ☐ Convert numbers in any given number base to any other base.
- ☐ Add and subtract binary numbers.
- ☐ Write binary numbers in their 8-bit two's complement format.
- ☐ Write binary numbers in hexadecimal.
- ☐ Use the BBC Micro to display numbers in decimal or hexadecimal.
- ☐ Read the contents of any memory location in the BBC Micro.
- ☐ Write either a decimal or a hexadecimal number into a memory location in the BBC Micro.

Answers to SAQs

SAQ 1

$$(a) 8352 = (8 * 10^3) + (3 * 10^2) + (5 * 10^1) + (2 * 10^0)$$

$$(b) 5032 = (5 * 10^3) + (0 * 10^2) + (3 * 10^1) + (2 * 10^0)$$

$$(c) 2100 = (2 * 10^3) + (1 * 10^2) + (0 * 10^1) + (0 * 10^0)$$

SAQ 2

(a) 4758 (b) 8063 (c) 5000

SAQ 3

$$(a) 1000_2 = 8 \quad (b) 1111_2 = 8 + 4 + 2 + 1 = 15$$

$$(c) 11111111_2 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

$$(d) 1001010_2 = 64 + 8 + 2 = 74$$

SAQ 4

(a)

	Remainder
$12/2 = 6$	0
$6/2 = 3$	0
$3/2 = 1$	1
$1/2 = 0$	1

$$12_{10} = 1100_2$$

(b)

	Remainder
$100/2 = 50$	0
$50/2 = 25$	0
$25/2 = 12$	1
$12/2 = 6$	0
$6/2 = 3$	0
$3/2 = 1$	1
$1/2 = 0$	1

$$100_{10} = 1100100_2$$

(c)

	Remainder
$255/2 = 127$	1
$127/2 = 63$	1
$63/2 = 31$	1
$31/2 = 15$	1
$15/2 = 7$	1
$7/2 = 3$	1
$3/2 = 1$	1
$1/2 = 0$	1

$$255_{10} = 11111111_2$$

We haven't written out the binary calculations for this one, but if you have reached the end, you should have found that

$$65535_{10} = 1111111111111111_2.$$

SAQ 5

(Carries and borrows are not shown.)

- (a)
$$\begin{array}{r} 110011 \\ + 1100 \\ \hline + 111111 \\ \hline \end{array} \begin{array}{l} (= 51) \\ (= 12) \\ (= 63) \end{array}$$
- (b)
$$\begin{array}{r} 1000 \\ + 101 \\ \hline 1101 \end{array} \begin{array}{l} (= 8) \\ (= 5) \\ (= 13) \end{array}$$
- (c)
$$\begin{array}{r} 110011 \\ + 1111 \\ \hline 1000010 \end{array} \begin{array}{l} (= 51) \\ (= 15) \\ (= 66) \end{array}$$
- (d)
$$\begin{array}{r} 11111111 \\ + 1 \\ \hline 100000000 \end{array} \begin{array}{l} (= 255) \\ (= 1) \\ (= 256) \end{array}$$
- (e)
$$\begin{array}{r} 1111 \\ - 110 \\ \hline 1001 \end{array} \begin{array}{l} (= 15) \\ (= 6) \\ (= 9) \end{array}$$
- (f)
$$\begin{array}{r} 1111 \\ - 101 \\ \hline 1010 \end{array} \begin{array}{l} (= 15) \\ (= 5) \\ (= 10) \end{array}$$
- (g)
$$\begin{array}{r} 110011 \\ - 1111 \\ \hline 100100 \end{array} \begin{array}{l} (= 51) \\ (= 15) \\ (= 36) \end{array}$$
- (h)
$$\begin{array}{r} 10000 \\ - 1 \\ \hline 1111 \end{array} \begin{array}{l} (= 16) \\ (= 1) \\ (= 15) \end{array}$$

SAQ 6

- (a) 00100101 (= 37)

Two's complement = 11011010+1 = 11011011 (= -37).

Verification:

$$\begin{array}{r} \text{Add} \quad 00100101 \\ \quad 11011011 \\ \hline 1 \quad 00000000 \end{array}$$

- (b) 00000001 (= 1)

Two's complement = 11111110+1 = 11111111 (= -1).

Verification:

$$\begin{array}{r} \text{Add} \quad 00000001 \\ \quad 11111111 \\ \hline 1 \quad 00000000 \end{array}$$

- (c) 00000000 (= 0)

Two's complement = 11111111+1 = 00000000 (= 0). (Ignore the carry from bit 7.)

No verification is needed but this example demonstrates that the two's complement method works for zero as well.

- (d) 01111111 (= 127)

Two's complement = 10000000+1 = 10000001 (= -127).

Verification:

$$\begin{array}{r} \text{add} \quad 01111111 \\ \quad 10000001 \\ \hline 1 \quad 00000000 \end{array}$$

SAQ 7

(a) $E9_{16} = 14 * 16 + 9 = 224 + 9 = 233_{10}$

(b) $10_{16} = 16_{10}$

(c) $100_{16} = 256_{10}$

(d) This can be done by a short cut, since $FFFF_{16}$ must be 1 less than 10000_{16} .

So $FFFF_{16} = 65536 - 1 = 65535$.

(e) $ABCD_{16} = 10 * 16^3 + 11 * 16^2 + 12 * 16 + 13$
 $= 10 * 4096 + 11 * 256 + 12 * 16 + 13$
 $= 40960 + 2816 + 192 + 13$
 $= 43981$

SAQ 8

(a) $0100\ 0100 = 44_{16}$

(b) $1000\ 0000 = 80_{16}$

(c) $0000\ 0001 = 01_{16}$ (or 1_{16})

(d) $1111\ 1111 = FF_{16}$

(e) $1010\ 1111 = AF_{16}$

SAQ 9

Every memory location is an 8 bit location holding 0's and 1's so the locations contain binary numbers from 00000000 to 11111111 (= 255 decimal).

SAQ 10

No answer is required.

UNIT 2

Addition and subtraction

- 2.1 The 6502 microprocessor
- 2.2 Loading the accumulator
- 2.3 Writing a program
- 2.4 STA
- 2.5 Labels
- 2.6 Addition
- 2.7 Subtraction
- 2.8 Adding larger numbers
- 2.9 Subtracting two-byte numbers
- 2.10 The program counter

Assignment B

Objectives of Unit 2

Answers to SAQs

2.1 The 6502 microprocessor

When you are programming in BASIC, you do not need to know what the computer does with the words and variables that you enter. But when you are programming in assembly language, you need to know something about the inner workings of the microprocessor used by your microcomputer. The 6502 is the microprocessor used by the PET, Apple and other micros. The BBC Micro uses the 6502A which is like the 6502 but runs at twice the speed.

The area of the 6502 that we are most concerned with in this course contains the six 'registers' – places in which numbers can be stored. Numbers which are not in these registers have to be kept outside the microprocessor in Random Access Memory (RAM). The registers of the 6502 are shown in Figure 2.1.

A register	(8-bit)	A
X register	(8-bit)	X
Y register	(8-bit)	Y
Stack	(8-bit)	S
Program counter	(16-bit)	PC
Processor status register	(8-bit) (Flags)	P

MEMORY

Not part
of the
microprocessor

Figure 2.1 The registers of the 6502 microprocessor

We shall not explain what each of these sections does until each is needed to solve a particular programming problem. In this unit we are going to look at the A register or accumulator.

The accumulator is an 8-bit register. It can hold a binary number between 00000000 and 11111111. This, as you have seen in Unit 1 is called a byte. As in the case of writing binary numbers, the register is always shown with the lsb (least significant bit) on the right and the msb (most significant bit) on the left.

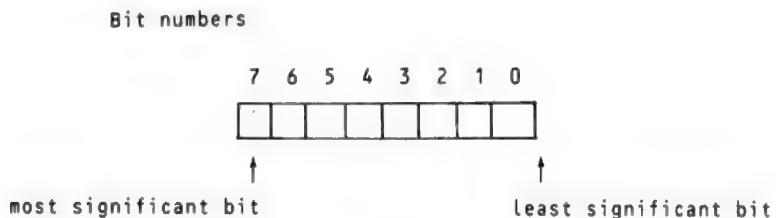


Figure 2.2 Bit numbers in the accumulator (A register)

The accumulator gets its name from the fact that it *accumulates* numbers. It is the part of the microprocessor where the arithmetic takes

place – or rather, appears to take place as far as the programmer is concerned. The arithmetic is actually performed by the Arithmetic Logical Unit (ALU) but the programmer works with the registers and not with the ALU. The accumulator is similar to the display on a pocket calculator although a calculator display does not perform calculations – the display is more like a 'window' which enables you to see into the calculator. When you add 5 and 3 with a pocket calculator you:

	Display of calculator
. put 5 into the memory	5
. press +	5
. put 3 into the memory	3
. press =	8

So all the numbers 5, 3 and 8 appear in the display. The accumulator in a microprocessor works in a somewhat similar way. Let's explore its behaviour in more detail.

2.2 Loading the accumulator

The accumulator is involved in most activities in the microprocessor. To put a number directly into the accumulator your program must execute the command:

LDA #number.

e.g.

LDA #5 puts decimal 5 into the accumulator

LDA #&3C puts &3C into the accumulator

LDA means Load Accumulator

This command is shown diagrammatically in Figure 2.3.



Figure 2.3 Action of LDA # 3C

immediate

In assembly language, numbers often refer to memory locations. If you want a number to be treated by your program as an ordinary numeric value you have to prefix it with the symbol '#'. This is read as 'immediate'.

(= immediate) means 'what follows is an ordinary number'

SAQ 1

What hex number is in the accumulator after each of the following?

- (a) LDA #3 (b) LDA #0 (c) LDA #255 (d) LDA #300 (e) LDA #&0
(f) LDA #&8 (g) LDA #&FF (h) LDA #&F00 .

2.3 Writing a program

So far we have only one assembly language command, but we will try to write a program using it. Assembly language programs in the BBC Micro can be written within BASIC. This is the method we are going to use in this course. The assembly part starts with a square bracket:

```
[  
and ends with  
RTS  
]
```

We need the RTS (ReTurn from Subroutine) at the end because we are going to execute the routine with CALL which treats the machine code as a sub-routine. We must therefore indicate when the subroutine has ended. When used in this way, RTS can be thought of as 'Return to BASIC'.

RTS = ReTurn from Subroutine

Here is our first assembly language program:

```
10 REM FIRST PROGRAM  
20  
30 DIM Z%50  
40 P% = Z%  
50 [  
60 LDA #&2A    \Put the number &2A into the accumulator  
70 RTS        \Return to BASIC  
80 ]  
90 END
```

Program 2.1

First, a few points about the program.

- ☐ It is meant to be an assembly language program but only lines 60 and 70 are assembly language statements. The symbol [announces the start of the assembly language part of the program and the symbol] declares the end of the assembly language. The rest is BASIC. This is a useful feature of the BASIC assembler provided in the BBC Micro. It makes it very easy to write and run assembly language programs.
- ☐ After each assembly language statement it is possible to insert comments to remind you of what is happening in the program. The

comments to remind you of what is happening in the program. The comment must be preceded by '`\`' in BBC assembly language. Most other assemblers use '`;`'. Some use a 'fixed format' where the statements and comments have to be typed into pre-defined columns on the screen.

- We have used one blank line to clarify the structure of the program. As our programs get longer we shall use more blank lines. In BBC BASIC, a blank line is created by typing the line number followed by one space. (Some programmers prefer to follow the line number with a '`;`' e.g.

20:

- The statement '`DIM Z%50`' has told the assembler to reserve 50 bytes of memory space immediately above the BASIC program. This space is labelled Z%. (Some assemblers use '`ORG`' to indicate the starting address of the program.) The assembler will place the machine code into this space (see below). The variable Z% is not special; you could use any numeric variable to label the space e.g. `DIM space%50`. If you use a floating point variable, then a space is needed between its name and the number of bytes e.g. `DIM Z 50` or `DIM space 50`. (Here we are reserving a block of space but leaving the assembler to place the machine code there. The reserved space can be addressed byte by byte using a technique described in section 6.8.)
- The '`!`' line is preceded by a line saying `P% = Z%`. P% is an 'assembler directive'. It tells the assembler where, in memory, to place the assembled program. You must use P% as the variable to which you assign the variable reserving the memory space. In Program 2.1 we could have replaced lines 10 and 20 by the single statement '`DIM P%50`'. This method is suitable only for programs that will contain only one assembly routine. If you have three routines in one program, each to be used at a different time, then you would reserve three separate blocks of memory e.g.

```
DIM A%50, B%75, C%30
```

You then set `P% = A%` before assembling the first routine, set `P% = B%` before assembling the second routine and set `P% = C%` before assembling the third routine.

Another method of placing the machine code in memory is to use a statement which directly assigns a memory location to P% e.g.

```
P% = &2000
```

This causes the assembler to put the machine code into memory locations &2000 upwards. You have to be careful with this method since the memory you use is not reserved and you have to make sure that nothing else (e.g. BASIC's variables) uses that space. So, the three approaches to reserving memory that we have suggested so far

are:	Method 1	Method 2	Method 3
	30 DIM Z%50	30 DIM P%50	30 P% = &2000
	40 P% = Z%	40 [etc	40 [etc
	50 [etc		

P% changes in value during the assembly process as the memory locations above its initial value are filled with machine code. So once a program is assembled, the value of P% is no longer what it was when the statement `P% = ...` was executed.

In 'fixed format' assemblers, horizontal spacing of the program is critical. This is not the case with the BBC BASIC assembler. All the programs in this course were typed in using `LIST0 0` (see *User Guide* page 290) without any space between line numbers and the statements that follow. We then told the computer to list them in `LIST0 1`. Towards the end of the course, we use `LIST0 7` to highlight indented loops.

The program looks fine – but how do we know if it works? As it stands, we can't find out if it works because we can't directly see what number is in the accumulator. Since you know how to read the contents of a memory location let's transfer the contents of the accumulator to a memory location. To do this we need a new command.

2.4 STA

STA means STore Accumulator

STA is not enough by itself to store the contents of the accumulator since we have to tell the computer where to store what is now in the accumulator. We have to specify a memory location. To start with, we'll use one of the locations between `&70` and `&8F`.

STA &70 means store the contents of the accumulator in memory location &70

'Store' actually means 'copy' rather than 'transfer'. After executing STA &70, the number that was in the accumulator is still there but is now also in memory location &70.

The action of this is shown in Figure 2.4.

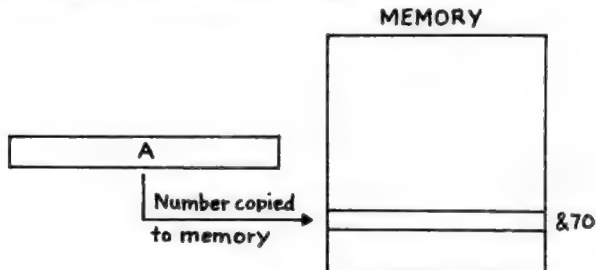


Figure 2.4 Action of STA &70

We shall now add this statement to Program 2.1 to make Program 2.2. At the same time we have changed `DIM Z%50` to `P% = &2000`. Either statement will provide the program with memory space but `P% = &2000` fixes the machine code at a precise location which makes it easier for you to compare what happens on your run with our run.

```

10 REM STORE A NUMBER
20
30 P% = &2000
40 [
50 LDA #&2A \Put the number &2A into the accumulator
60 STA &70  \Store the contents of the accumulator in memory
              \location &70
70 RTS      \Return to BASIC
80 ]
90 END

```

Program 2.2

[K] Before you run your program, put 0 into memory location &70 by typing `?&70 = 0`. Then run the program by typing `RUN` in the same way as you would for an ordinary BASIC program. Check that the following appears:

```

>RUN
2000
2000 A9 2A    LDA #&2A \Put the number &2A into the accumulator
2002 85 70    STA &70  \Store the contents of the accumulator
                  in memory location &70
2004 60       RTS      \Return to BASIC

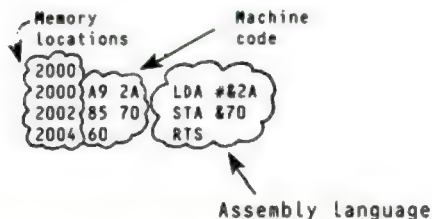
```

Program 2.2 (assembled)

We shall return to what all this means in a moment; but first, what has it done? Check the contents of &70 by typing `P. ?&70`. It is still 0! The program has been 'RUN' but has not done what we expected it to do. That is because an assembly program involves two processes: `RUN` and `CALL`.

RUN and CALL

Typing `RUN` as above assembles the program. That is, it turns the assembly listing that we typed in into machine code. The machine code created from our listing is shown below.



Once the program is assembled, you use it by a `CALL` command from BASIC. You can use `CALL` in three ways:

- ❑ CALL the name of the memory location at which the program starts.
To do this, type

CALL Z%

if you used Z% as the label for the memory space for the program.

- ❑ CALL the number of the memory location if you used P% = &2000 i.e

CALL &2000

- ❑ CALL a 'label' attached to the start of the program. This will be explained in section 2.5.

(Remember that since the value of P% changes during the assembly process, typing CALL P% won't work.)

We shall now try all three of these.

Method 1 CALL memory location

- ❑ Type RUN to assemble Program 2.2.
Then type CALL &2000.
Now type P. ?&70.

You should see the number 42 on the screen since the program put &2A (= 42) into &70. Also type P. ~?&70 to see the result in hex.

Here is how our test run appears on the screen:

Test run

>?&70 = 0	Put the number 0 into memory location &70 at start.
>CALL &2000	Call the program.
>P. ?&70	Check the contents of memory location &70.
42	It is now 42 as we would expect.
>P. ~?&70	Check again but in hex
2A	

Method 2 CALL Z%

To check this method, change Program 2.2 to:

```
10 REM STORE A NUMBER
20
30 DIM ZX$0
40 PX = ZX
50 [
60 LDA #&2A \Put the number &2A into the accumulator
65 STA &70 \Store the contents of the accumulator in memory
        \location &70
70 RTS \Return to BASIC
80 ]
90 END
```

and assemble it.

```

>RUN
0EE0 A9 2A    LDA #&2A
0EE2 85 70    STA &70
0EE4 60       RTS

```

Notice that the machine code is just the same as before but the memory locations are different because DIM Z% has placed the machine code above the BASIC program. (Your locations may be slightly different since the start location is now determined by the length of the BASIC program including the comments and any spaces you have typed in.)

To execute the program, use CALL Z% or CALL &EEO:

```

>?&70 = 0    Re-set location &70 to 0 so that we can test what happens
>CALL Z%
>P."?&70
      2A

```

Method 3 Calling a label

This is described in section 2.5.

SAQ 2

In Program 2.2, we loaded the accumulator in 'immediate mode' i.e. we put the number &2A directly into it. Write an assembly program which loads the accumulator with the contents of &70 and stores the result at &71. Check that your program works properly by putting the number &2D into memory location &70 before the run.

SAQ 2 demonstrates a very important point about 6502 assembly language: numbers cannot be transferred directly from one memory location to another. To copy from one memory location to another, you have to first load the number from one memory location into a register and then copy it from the register into the second memory location.

2.5 Labels

A BBC assembly language label goes at the beginning of a line and starts with a '.'. For Program 2.2 we need a label at the beginning of the program so we shall call it 'start'. Program 2.3 shows it inserted into Program 2.2 at line 25.

```

10 REM STORE A NUMBER
20
30 PX = &2000
40 [
50 .start
60 LDA #&2A    \Put the number &2A into the accumulator
70 STA &70     \Store the contents of the accumulator in memory
               location &70
80 RTS        \Return to BASIC
90 ]
100 END

```

Program 2.3

☒ Check that it works as follows:

- ☐ Reset `&70` to 0 with `?&70 = 0`.
- ☐ Type `RUN`.
- ☐ Type `CALL start`. (Without the `.` which appears in front of the label in the program.)
- ☐ Check the contents of `&70` with `P.~?&70`.

You should get the answer 2A as before:

Test run

```
>?&70 = 0
>CALL start
>P.~?&70
    2A
```

In Program 2.3 we put the label on a new line above the position it referred to (`LDA #&2A`). Some programmers like to put the label on the same line as the item it refers to. In this case we would have written

```
60 .start LDA #&2A
```

If you try both methods, you'll see that the memory locations shown in the left-hand column during the assembly are identical.

Warning: These labels are part of BASIC. If you use a word both as a BASIC variable and as an assembly label, or use the same word for two different labels, you will not get an error message. The BASIC just re-assigns the variable to the latest use that you have specified and your assembled program will be faulty.

2.6 Addition

So far, we have loaded a number into the accumulator but we haven't done any calculations on the number. One thing that we can use the accumulator for is adding. Numbers are added to the contents of the accumulator with the command

ADC ADd with Carry

The ADC command adds whatever follows it to whatever number is in the accumulator. At this stage let's look at two items that might follow ADC.

`ADC #&30` means add the *number* `&30` to the current contents of the accumulator.

`ADC &70` means add the *contents* of the memory location `&70` to the current contents of the accumulator.

The steps involved in carrying out `ADC &70` are shown in in Figure 2.5. The operation uses the arithmetic logical unit (ALU) of the microprocessor. This is where the microprocessor carries out its arithmetic and logical operations. Whenever an operation is performed by the

ALU, the result is always left in the accumulator which is itself part of the ALU.

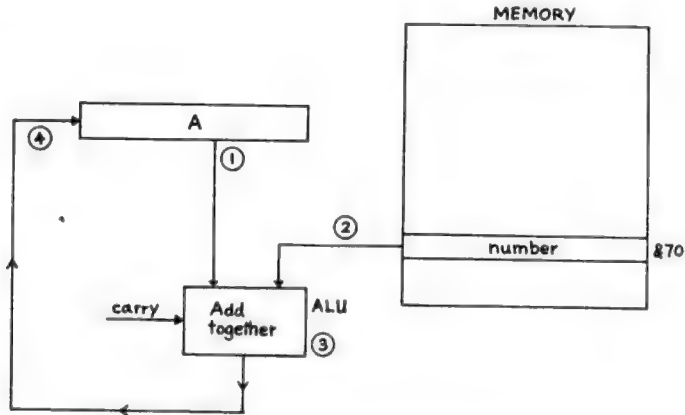


Figure 2.5 Action of ADC &70

These steps are

- ☐ Step 1 The first number is already in the accumulator.
- ☐ Step 2 Copy the number in memory location &70 into the ALU.
- ☐ Step 3 Add the two numbers together. Add any carry in at the same time (see below).
- ☐ Step 4 Copy the result in the accumulator.

Notice that the result of ADC, whatever it is followed by, is left in the accumulator.

ADC is quite straightforward apart from one small point. Because the accumulator can only hold numbers up to &FF we might well get a carry when two numbers are added. The microprocessor detects this carry with a 'flag' that we will explain later in the course. For now, we shall just say that ADC is sometimes preceded by an instruction to clear the carry flag. You will need to use this instruction until we discuss when it can be omitted.

`CLC` CLear Carry: sets the carry flag to 0

We can now write a program to add two numbers. Let's add &25 and &30. The program is straightforward. However, it uses one new idea, which is to get the BASIC program itself to CALL the assembled machine code so that we don't have to do this at the keyboard. In Program 2.4, the assembly language of lines 40 to 110 will have been assembled by the time the program executes line 110. So, at line 150 we

can CALL the machine code by executing CALL begin.

```
10 REM ADDITIONS
20
30 PX = &2000
40 [
50 .begin
60 LDA #&25      \Put the number &25 into the accumulator
70 CLC          \Clear carry before the addition
80 ADC #&30      \Add the number &30 to the contents of the accumulator
90 STA &72       \Store the result in memory location &72
100 RTS
110 ]
120
130 REM Test routine
140
150 CALL begin
160
170 REM Print out result
180 PRINT "Contents of location &72 = &" ; "&&72"
```

Test run

>RUN

Contents of location &72 = &55

Program 2.4

 Program 2.4.

SAQ 3

Repeat Program 2.4 with the following numbers in lines 60 and 80.

(a) 16 and 0 (b) &40 and &30 (c) &40 and &F0

SAQ 4

Write a program to store the numbers &30 and &25 in memory locations &70 and &71; take the numbers from those locations and add them together; store the result in &72. Check your program by assembling and calling it.

2.7 Subtraction

Provided that we avoid the problem of what happens when the result is negative, subtraction is almost identical to addition. The statement

SBC SuBtract with Carry

subtracts whatever follows it from the current contents of the accumulator. As with ADC, we'll consider two cases at this stage.

SBC #&30 means subtract the number &30 from the current contents of the accumulator.

SBC &70 means subtract the contents of &70 from the current contents of the accumulator.

The steps involved in carrying out SBC &70 are shown in Figure 2.6.

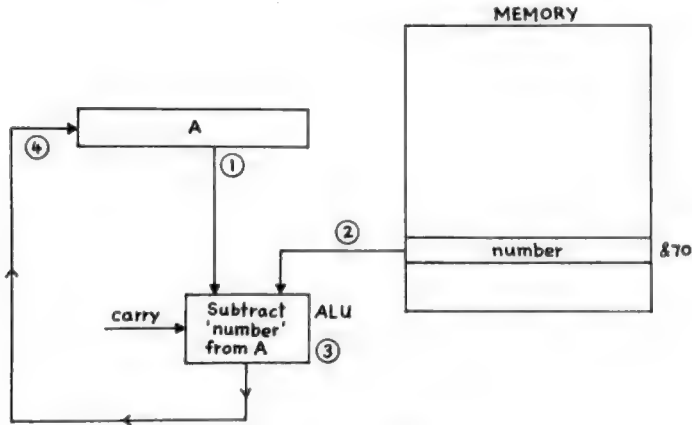


Figure 2.6 Action of SBC &70

The steps are:

- ❑ Step 1 The first number is already in the accumulator.
 - ❑ Step 2 Copy the number in memory location &70 into the ALU.
 - ❑ Step 3 Subtract the two numbers with a borrow if necessary.
 - ❑ Step 4 Copy the result back into the accumulator.
- Again, as with ADC, the result is left in the accumulator.

Setting the carry flag

With ADC we had to clear the carry flag (i.e. make it 0) before adding. With SBC we have to set the carry flag (i.e. make it 1) before we use SBC. This is done with SEC.

(Note: 6502 subtraction using the SBC command works differently from subtraction commands in many other microprocessor languages).

SEC means SET Carry flag

We shall leave the explanation of why this is necessary, for a short while, and use SEC according to this rule. The following program is a

simple subtraction program to subtract the contents of location &70 from the number &38.

```
10 REM SUBTRACTION
20
30 P% = &2000
40 [
50 .start
60 LDA #&38 \Put the number &38 into the accumulator
70 SEC \Set the carry flag
80 SBC &70 \Subtract the contents of memory location &70
           from the accumulator
90 STA &71 \Store the result in memory location &71
100 RTS
110 ]
120
130 REM Test routine
140
150 REM Test data
160 ?&70 = &14
170
180 CALL start
190
200 REM Print out result
210 PRINT "Contents of location &71 = &" ; "?&71"
```

Test run

```
>RUN
2000
2000          .start
2000 A9 38     LDA #&38 \Put the number &38 into the accumulator
2002 38       SEC \Set the carry flag
2003 E5 70     SBC &70 \Subtract the contents of memory location
                &70 from the accumulator
2005 85 71     STA &71 \Store the result in memory location &71
2007 60       RTS
```

Contents of location &71 = &24

Program 2.5

☒ Set the contents of &71 to 0 and put a number of your choice into &70. Then RUN Program 2.5.

SAQ 5

Write a program to subtract the number in memory location &71 from the number in memory location &70. Store the result in a location &72. Put suitable numbers (i.e. between &0 and &FF and with the number in location &70 greater or equal to that in &71) in the first two locations and RUN and CALL your program.

2.8 Adding larger numbers

Program 2.6 is a typical program for adding two *one byte* numbers:

```
10 REM ADD TWO NUMBERS
20
30 DIM ZX50
40 PX = ZX
50 [
60 .add
70 LDA &70    \Put the first number in the accumulator
80 CLC        \Clear the carry flag
90 ADC &71     \Add the second number to the number in the
               accumulator
100 STA &72    \Store the result in memory location &72
110 RTS
120 ]
130
140 REM Test routine
150
160 INPUT "Enter first number (decimal) " first
170 INPUT "Enter second number (decimal) " second
180
190 ?&70 = first
200 ?&71 = second
210
220 CALL add
230
240 REM Print out result
250 PRINT "Contents of location &72 = &" ; "?&72

>RUN
Enter first number (decimal) 45
Enter second number (decimal) 12
Contents of location &72 = &39
```

Program 2.6

If you use this program to add together different numbers, some results look right whilst others do not e.g.

&70 (first number)	&23	&D5	&F4
&71 (second number)	&45	&34	&23
&72 (result)	&68	&9	&17

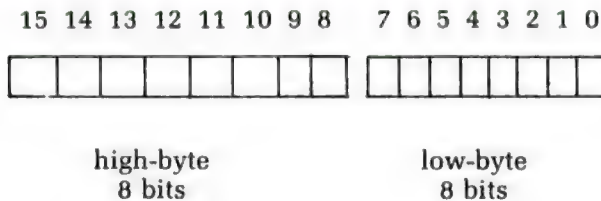
The first answer is correct ($&23 + &45 = &68$) but what about the other two?

SAQ 6

Why are the other two answers wrong? How could we make them correct?

Clearly, then, Program 2.6 is of limited use. It is only safe to use as it stands provided we are certain that the answer will not exceed &FF (255). If the answer exceeds that value, then the carry will be lost. If we want to add numbers in circumstances where the answer could exceed &FF then we need a better program.

Before we develop that program, we must consider how the computer will store these larger numbers. Any number larger than &FF will require more than 8 bits. We can't store, say, 343 in &70. The solution is to store the numbers as two bytes (or three or more for even larger numbers). If we store all our numbers as two bytes, each consisting of a low-byte and a high-byte then the format will be:



The smallest (positive) number that can be stored in this format is

$$0000000000000000 = 0.$$

The largest number is

$$1111111111111111 = 2^{16} - 1 = 65535.$$

If we are prepared to represent numbers as two bytes, we can deal with quite large numbers. But the cost of this is that each number requires twice as much memory and we need more complicated programming to handle the numbers.

SAQ 7

How would you store these numbers in memory if all numbers to be stored will use two bytes? (Use memory locations &70 and &71.)

(a) &F3A2 (b) &F200 (c) &32

Two-byte numbers

In 6502 machine code programs, two-byte numbers are always stored with the low-byte in the first (lower) memory location and the high-byte in the memory location above it.

Consider what happens when you add two two-byte numbers. There are two cases that can arise:

	High byte	Low byte		High byte	Low byte
(a)	&2E	D5	(b)	&25	10
	&10	61		&13	4F

SAQ 8

What are the answers to these two problems and what is the difference between them?

We have now established that a carry can arise between the low- and high-byte additions. Why should this matter? It matters because the accumulator is only 1 byte long, so the two two-byte numbers have to be added byte by byte: first add the two low-bytes to each other; then add the two high-bytes to each other. But, if we are not careful, we will lose the carry.

Fortunately the 6502 helps us here since it looks after the carry. All we have to do is to *not* use CLC before the high-byte addition. This is illustrated in the flowchart for two-byte addition shown in Figure 2.7. (Remember that ADC means 'Add with Carry').

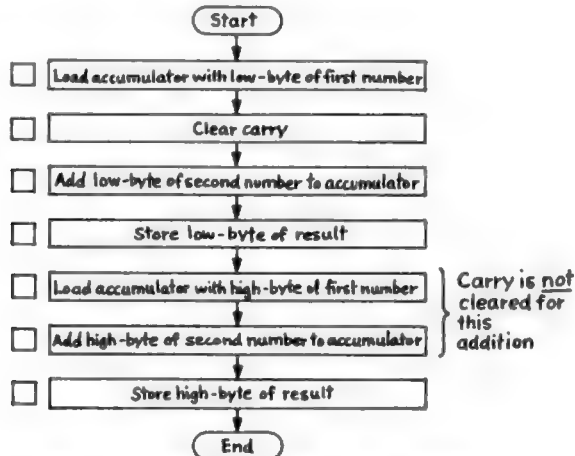


Figure 2.7 Adding two two-byte numbers

SAQ 9

Fill in the boxes at the side of Figure 2.7 with the hex numbers resulting from each operation for the addition of &2ED5 and &1061.

Program to add two two-byte numbers

We can now use Figure 2.7 to develop a program. The numbers will be stored as in Figure 2.8.

First number	D5	MEMORY
	2E	&70
Second number	61	&71
	10	&72
	result: low-byte	&73
	result: high-byte	&74
		&75

The program for two-byte addition is shown in Program 2.7.

```

10 REM ADDING TWO-BYTE NUMBERS
20
30 DIM Z%50
40 P% = Z%
50 [
60 .two_byte_add
70 LDA &70 \Load low-byte of first number into accumulator
80 CLC \Clear carry for low-byte addition
90 ADC &72 \Add low-byte of second number to contents of
\accumulator
100 STA &74 \Store low-byte of result
110 LDA &71 \Load high-byte of first number into accumulator
120 ADC &73 \Add high-byte of second number, plus any carry,
\to contents of accumulator
130 STA &75 \Store high-byte of result
140 RTS
150 ]
160
170 REM Test routine
180
190 REM Enter test data
200 ?&70 = &D5
210 ?&71 = &2E
220 ?&72 = &61
230 ?&73 = &10
240
250 CALL two_byte_add
260
270 REM Print out result
280 PRINT "Contents of location &74 (= low-byte) = &" ; "&74
290 PRINT "Contents of location &75 (= high-byte) = &" ; "&75

```

Program 2.7

Test run

```

>RUN
Contents of location &74 (= low-byte) = &36
Contents of location &75 (= high-byte) = &2F

```

SAQ 10

Confirm that the answer is correct for the above run by doing the sum on paper.

We have now shown how CLC works and why it needs clearing *before* some additions and *leaving alone* before others. If CLC is needed before an addition, where you put the CLC is up to you as long as it appears before the addition and after anything else that could affect it. We believe that programs are clearest if the CLC comes immediately before the relevant ADC statement and that is the convention we shall use.

This can be summarised as follows:

CLC

Use CLC before an addition with ADC into which any carry must not be added.

Omit CLC before an addition with ADC into which any carry must be added.

2.9 Subtracting two-byte numbers

You have probably guessed by now what we have to do in two-byte subtraction. The problem to overcome is that we could have a borrow between the low and high bytes. Just as the 6502 took care of the carry in two-byte addition, so it takes care of the borrow in two-byte subtraction. All we have to do is to *not* set the carry with SEC before the high-byte subtraction. This is probably easier to understand by reading SEC as 'clear the borrow'.

SAQ 11

Modify Program 2.7 to make it subtract two two-byte numbers. Run and check that your program correctly subtracts &28A3 from &5D2A. Repeat for &C47B and &93.

SAQ 12

Modify your answer to SAQ 11 to subtract a one-byte number from a two-byte number. Check that your program works by subtracting &3A from &8CF5.

And finally, a reminder about using SEC in subtractions.

SEC

Use SEC before a subtraction with SBC which should not include any borrow.

Omit SEC before a subtraction with SBC which should include any borrow.

2.10 The program counter

Now that we have run and assembled some machine code programs, we can take a brief look at the 'program counter'. This is one of the six 6502 registers and is used in all 6502 operations. It is mentioned here not because of any special connection with addition and subtraction but because you now have some material on which to observe its activities.

The program counter always contains the 'address' (i.e. position in memory) of the next instruction to be executed. Thus if you look at the assembled version of Program 2.5 you can see, for example:

```
2000 A9 38 LDA #38
```

This tells us that the assembler converted `LDA #38` into the machine code instruction `A9 38` which it stored at `&2000`. When executing this instruction, the program counter would contain `&2002`, the address of the next instruction.

If you look down the first column of the assembled program, you will see that different instructions require different amounts of memory. Thus `LDA #38` used `&2000` to `&2001` (two bytes) and `SEC` used `&2002` (one byte).

The assembler works this memory allocation out for you so we shall not be saying much more about the program counter in this course.

P%

During an assembly run on the BBC Micro, the variable `P%` contains the current value of the program counter. Hence the statement

```
P% = Z%
```

at entry to an assembly program. The program counter is loaded with the value at the memory location (`Z%`) which is the start of the assembly listing.

`P%` must not be declared as 'LOCAL' to a procedure or subroutine containing your assembly language program.

The decimal mode

In this unit we have looked at the 6502 performing arithmetic calculations on binary numbers. The 6502 can be switched into a mode which allows it to operate on binary-coded decimal numbers but we have not described this feature in this course.

Assignment B

1. Write a program to add two three-byte numbers. Check that it works correctly by testing with suitable data. What is the largest number that can result from this program?

Objectives of Unit 2

After studying this unit you should be able to:

- ☐ Identify the bit numbers in a binary number.
- ☐ Use LDA to put a number into the accumulator (directly or from memory).
- ☐ Use # to indicate 'a number follows'.
- ☐ Use [] to indicate the start and end of an assembly language program.
- ☐ Use RTS to return to BASIC.
- ☐ Use DIM ZX to reserve memory space for an assembly program.
- ☐ Use PX to place an assembly program in reserved space.
- ☐ Use STA to store the contents of the accumulator.
- ☐ Assemble a program into machine code.
- ☐ CALL a machine code program.
- ☐ Use a label to indicate the start of a program.
- ☐ Insert comments into an assembly program.
- ☐ Use ADC with and without CLC as appropriate to add numbers.
- ☐ Use SBC with and without SEC as appropriate to subtract numbers.
- ☐ Write an addition program for two-byte numbers.
- ☐ Write a subtraction program for two-byte numbers.

Answers to SAQs

SAQ 1

(a) &03 (b) &00 (c) &FF (d) You can't load the accumulator with 300 (&12C) since it is larger than 8 bits. Attempting to do so produces an error message ('Byte') from the assembler. (e) &00 (f) &08 (g) &FF (h) Answer as for (d)

SAQ 2

```
10 REM LOAD AND STORE
20
30 DIM ZX50
40 PX = ZX
50 [
60 LDA &70 \Load the accumulator with contents of memory
           \location &70
70 STA &71 \Store the contents of the accumulator in
           \memory location &71
80 RTS
90 ]
```

```

100
110 REM Test routine
120
130 REM Test data
140 ?&70 = &2D
150
160 CALL Z%
170
180 REM Print out result
190 PRINT "Contents of location &71 = &" ; "?&71

```

Program 2.8

Test run

```

>RUN
Contents of location &71 = &2D

```

In effect, this program uses the accumulator to transfer the contents of one memory location to another memory location.

SAQ 3

Your results (in hex) should be:

(a) &10 (b) &70 (c) &30 This last result is 'correct' in one sense. The addition was $\&40 + \&F0 = \&130$. Since the accumulator can only hold one one-byte number, the '1' in &130 was a carry out of the accumulator. This left &30 in the accumulator.

SAQ 4

```

10 REM ADDITION OF &30 and &25
20 REM Not an efficient way of doing the job
30
40 DIM Z%50
50 P% = Z%
60 [
70 .start
80 LDA #&30      \Put first number into accumulator
90 STA &70       \Store first number in memory location &70
100 LDA #&25     \Put second number into accumulator
110 STA &71      \Store second number in memory location &71
120 CLC         \Prepare to add (clear carry)
130 ADC &70     \Add first number to contents of accumulator
140 STA &72     \Store the result in memory location &72
150 RTS
160 ]
170
180 REM Test run
190
200 CALL start
210 PRINT "Contents of location &72 after addition
      = &" ; "?&72

```

Test run

```
>RUN
Contents of location &72 after addition = &55
```

Program 2.9

This program is not a demonstration of how to add &30 to &25. What it does emphasise is that to put numbers into memory, we have to first put them into the accumulator (or one of the other registers that we will introduce later) and then store the contents of the accumulator in the chosen memory location.

SAQ 5

```
10 REM SUBTRACTION OF ONE-BYTE NUMBERS
20
30 DIM Z%50
40 P% = Z%
50 [
60 .start
70 LDA &70 \Put number from memory location &70 into
           accumulator
80 SEC \Set carry flag before subtraction
90 SBC &71 \Subtract number in memory location &71 from
           contents of accumulator
100 STA &72 \Store the result in memory location &72
110 RTS
120 ]
130
140 REM Test run
150
160 REM Test data
170 ?&70 = &9E
180 ?&71 = &1F
190
200 CALL start
210
220 PRINT "Contents of location &72 = &" ; "&72"
```

Program 2.10

Test run

```
>RUN
Contents of location &72 = &7F
```

Test run

```
>RUN
Contents of location $72 = $7F
```

Program 2.10

SAQ 6

In both cases, the answer exceeds &FF and a carry occurs. The carry is lost in this program so the number in the accumulator is &100 short of the correct answer i.e.

```
&D5 + &34 = &109
lost carry = &100
balance   = &9 (left in accumulator)
```

SAQ 7

	MEMORY	
(a)	A2	&70
	F3	&71
(b)	00	&70
	F2	&71
(c)	32	&70
	00	&71

Even though &32 only requires one byte, it has to be stored as &0032 to fit two bytes. A program using two-byte numbers must store all numbers as two bytes even though some numbers will be less than &100.

SAQ 8

(a) &3F36 (b) &385F

The difference between the two cases is that (a) generates a carry between the low- and high-byte additions whereas (b) does not.

SAQ 9

```
&D5 in accumulator
&36 in accumulator (the carry is in the carry flag)
&36 in memory
&2E in accumulator
&3F in accumulator (&2E + &10 + carry)
```

SAQ 10

```

      &2ED5
      + &1061
      -----
      &3F36
    /      \
high-byte = &3F  low-byte = &36
= contents of &75 = contents of &74
```

SAQ 11

```
10 REM SUBTRACTION OF TWO-BYTE NUMBERS
20
30 DIM Z%50
40 PX = Z%
50 [
60 .sub_2
70 LDA &70      \Load low-byte of first number into
                  accumulator
80 SEC          \Set carry for low-byte subtraction
90 SBC &72      \Subtract low-byte of second number from
                  accumulator
100 STA &74      \Store low-byte of result
110 LDA &71      \Load high-byte of first number into
                  accumulator
120 SBC &73      \Subtract high-byte of second number from
                  accumulator
130 STA &75      \Store high-byte of result
140 RTS
150 ]
160
170 REM Test run
180
190 REM Test data
200 ?&70 = &2A
210 ?&71 = &5D
220 ?&72 = &A3
230 ?&73 = &28
240
250 CALL sub_2
260
270 PRINT "Contents of location &74 (result low-byte) = &" ;
    ~?&74
280 PRINT "Contents of location &75 (result high-byte) = &" ;
    ~?&75
```

Program 2.11

Test run (first case)

```
>RUN
Contents of location &74 (result low-byte) = &87
Contents of location &75 (result high-byte) = &34
```

Test run (second case)

Set the test data in lines 200 to 230 to ?&70 = &7B, ?&71 = &C4, ?&72 = &93 and ?&73 = 0 (don't forget that although &93 is a one byte number, it still has to be entered as the two-byte number &0093). You should find that, after running Program 2.11, &74 contains &E8 and &75 contains &C3.

SAQ 12

```
10 REM SUBTRACTING ONE BYTE FROM TWO BYTES
20
30 DIM Z%50
40 P% = Z%
50 [
60 .sub_2
70 LDA &70      \Load low-byte of first number into
                  accumulator
80 SEC          \Set carry for low-byte subtraction
90 SBC &72      \Subtract low-byte of second number from
                  accumulator
100 STA &74      \Store low-byte of result
110 LDA &71      \Load high-byte of first number into
                  accumulator
120 SBC #0       \Subtract 0 as high-byte of second number
130 STA &75      \Store high-byte of result
140 RTS
150 ]
160
170 REM Test run
180
190 REM Test data
200 ?&70 = &F5
210 ?&71 = &8C
220 ?&72 = &3A
230
240 CALL sub_2
250
260 PRINT "Contents of location &74 (result low-byte) = &" ;
    "~?&74
270 PRINT "Contents of location &75 (result high-byte) = &" ;
    "~?&75
```

Program 2.12

Test run

```
>RUN
Contents of location &74 (result low-byte) = &BB
Contents of location &75 (result high-byte) = &8C
```

UNIT 3

Jumps, loops and branches

3.1 Flags

3.2 The zero flag

3.3 Conditional branches 1

3.4 The assembly process and OPT

3.5 Conditional branches 2

3.6 Subroutines

3.7 The X and Y registers

3.8 Unconditional jumps

3.9 Compare

Assignment C

Objectives of Unit 3

Answers to SAQs

3.1 Flags

In Figure 2.1 of Unit 2 we showed the six registers of the 6502. One of these was simply labelled 'flags'. We will now take a closer look at this register and at the action of some of the flags. The register is an 8-bit register in which each bit provides information about what is happening in the microprocessor. The register is shown in more detail in Figure 3.1.



Figure 3.1 The processor status (flags) register

In this unit we are going to need the carry flag (C) and the zero flag (Z). We shall also touch on the negative flag (N). Since the other flags are not needed at this stage, we shall not go into any detail about them.

Certain flags are affected by certain 6502 instructions. However, some instructions have no effect on the flags at all. Even though you have not met many instructions, we will list those instructions which have no effect on any of the flags. (The list will be useful for reference later).

The branch instructions

- BCC
- BCS
- BEQ
- BMI
- BNE
- BPL
- BVC
- BVS

The jump instructions

- JMP
- JSR

The 'do nothing' instruction

- NOP

Return from subroutine

- RTS

The store instructions

STA
STX
STY

And some rather special instructions

PHA
PHP
TXS

The carry flag in addition

We have – in effect – already noted the action of the carry flag. If an addition generates a carry (i.e. if the result is greater than &FF) then the carry flag is set to 1 by the microprocessor. The statement CLC sets the carry flag to 0.

CLC means CLear Carry
It sets the carry flag to 0

We can now deal with the use of CLC in more detail than in Unit 2. When you are in the middle of a program you might not know what value the carry flag has. If it is 1, then ADC will add the number in the accumulator to the number specified in the ADC statement plus 1 i.e. the carry. Hence the name 'Add with carry'. You can think of it as being like 'Add with 1 if the carry flag is 1'. But if you are adding two low-bytes, there should not be a carry in the addition. So you precede ADC low-byte addition with CLC to make sure that the carry flag is 0. In Program 2.7 we had:

```
10 REM ADDITION OF TWO-BYTE NUMBERS
20
30 DIM Z%50
40 PX = Z%
50 [
60 .two_byte_add
70 LDA &70          \Load low-byte of first number into
                    accumulator
80 CLC              \Clear carry for low-byte addition
90 ADC &72          \Add low-byte of second number to contents
                    of the accumulator

etc.
```

The CLC at line 80 made the carry flag equal to 0 so that no carry would be added into the low-byte addition. But the result of this addition *might* generate a carry to add into the high-byte addition. If it does, the carry flag will go to 1 at line 90. We must not clear this value since it will be added into the high-byte addition by the 6502 in the next part of the program.

100 STA &74	\Store low-byte of result
110 LDA &71	\Load high-byte of first number into accumulator
120 ADC &73	\Add high-byte of second number plus any carry to accumulator
130 STA &75	\Store high-byte of result
140 RTS	
150]	

SAQ 1

(a) What would be the effect of not clearing the carry flag before the start of the two-byte addition program?

(b) What would be the effect of clearing the carry flag between lines 90 and 100 of the two-byte addition program?

The carry flag in subtraction

The carry flag is also affected by SBC and takes the *complement* of the carry flag away from the accumulator. So if the carry is 1, 0 is taken off the accumulator. If the carry is 0, 1 is taken off the accumulator. This effects a borrow whenever it is required.

So, SEC (SEt Carry to 1) is needed at the *start* of a subtraction to prevent a borrow taking place. SEC is *not* used between a low- and high-byte subtraction since a borrow must be allowed to happen if it is needed as a result of the low-byte subtraction.

SAQ 2

In Program 3.1, what would be the value of the carry flag at lines 80 and 90 when the numbers being subtracted are:

(a) &9F53 - &2561 (b) &4DA2 - &1077?

10 REM SUBTRACTION	
20	
30 DIM Z%50	
40 P% = Z%	
50 [
60 .sub2byte	
70 LDA &70	\Low-byte of first number into accumulator
80 SEC	\Set carry for low-byte subtraction
90 SBC &72	\Subtract low-byte of second number from accumulator
100 STA &74	\Store low-byte of result
110 LDA &71	\High-byte of first number into accumulator
120 SBC &73	\Subtract high-byte of second number from accumulator
130 STA &75	\Store high-byte of result
140 RTS	
150]	
160 END	

3.2 The zero flag

The zero flag → **Z**

You have seen how to set or clear the carry flag. The zero flag is different because there is no specific instruction which will set or clear it. The zero flag is set and cleared *automatically* by the 6502. Its job is to tell you whether or not the result of the *last* operation that could have affected the Z flag was 0 or not. (Thirty of the fifty-six 6502 instructions affect the Z flag – see Appendix 2 for details).

Z = 1 means the last result was 0
Z = 0 means the last result was not 0

Be careful about Z being 0 when the result was *not* zero. Flags tell us whether a condition is true or not. In microprocessor logic, 1 = 'true' and 0 = 'false'. The condition we are testing is 'was the result 0?'. If it was (i.e. if true) then Z = 1 to indicate 'true'.

SAQ 3

What value will Z be at the end of each of the following routines?

- (a) LDA #&00
- (b) LDA #&8A
SEC
SBC #&53
- (c) LDA #&8A
SEC
SBC #&8A

3.3 Conditional branches 1

We can now introduce the first two branching statements for the 6502. You will have met the idea of a branch in BASIC and be aware that there are two types of branch:

Conditional branch: IF... THEN (line number)
Unconditional branch: GOTO (line number)

Both types of branch are available in 6502 assembly language and there are eight conditional branches. Why eight when one is good enough in BASIC? The reason is that the BASIC conditional branch, IF ... THEN ..., allows us to put any syntactically correct condition between IF and THEN so the one conditional branch statement can evaluate any valid condition that we require. But assembly language branches can only take place according to the settings of the various flags.

Two conditional branch statements depend on the state of the zero flag:

BEQ	Branch if EQual to zero (i.e. if Z = 1)
BNE	Branch if Not Equal to zero (i.e. if Z = 0)

The branch will cause the program to continue at some other instruction in the program. You can therefore branch to a memory location (as listed in the program counter line) or to a label in the program. Branching to a memory location is asking for trouble since you may get the branch wrong. We shall concentrate on branching to labels. First we shall look at what happens as soon as we introduce a very simple branch into a program.

The program we are going to look at examines the contents of &70. If the content of the address &70 is 0 then the run ends. If it does not contain zero then the program puts the number &FF into location &71. A flowchart for this is shown in Figure 3.2.

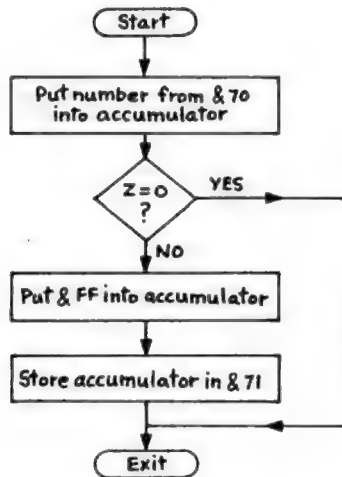


Figure 3.2 Flowchart showing use of BEQ

Turning this into a program produces Program 3.2.

```
10 REM BEQ
20
30 P% = &2000
40 [
50 .start
60 LDA #0
70 STA &71      \Clear indicator location at start
80 LDA &70      \Put contents of &70 into accumulator
90 BEQ exit     \End run if location &70 contained zero
```

```

100 LDA #&FF      \Put the number &FF into accumulator as
                  branch indicator
110 STA &71        \Store indicator at memory location &71
120 .exit
130 RTS
140 J
150 END

```

Program 3.2


The branch occurs at line 90 with the statement

BEQ exit.

This tells the program to branch to the statement immediately following the label 'exit' if the result of the last operation was zero. Since the previous operation (line 80) was

LDA &70

the branch tests whether or not the number in the accumulator is 0. As we loaded the accumulator from the memory location &70, we are, in effect, testing the contents of &70.

 Type and run Program 3.2.

The result is a bit disappointing. The program looks alright but the computer tells us that it doesn't like the program and responds with the message 'No such variable at line 90'.

```

>RUN
2000
2000      .start
2000 A9 00  LDA #0
2002 85 71  STA &71      \Clear indicator location at start
2004 A5 70  LDA &70      \Put contents of &70 into accumulator
No such variable at line 90

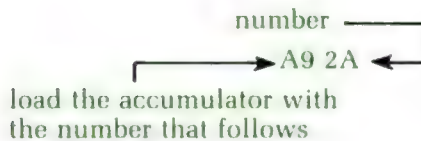
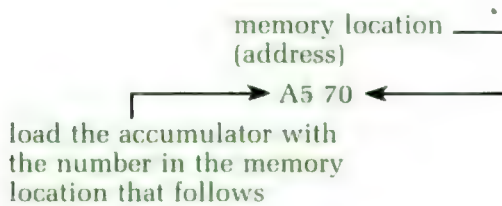
```

To explain why this has happened we must break off and look at what the assembler is doing. We then need to introduce a new statement, OPT.

3.4 The assembly process and OPT

When we type RUN, the computer *assembles* a machine code listing of the assembly language program that we have written. The computer cannot execute an *assembly listing*. Assembly language is a means by which we can write machine code programs without having to understand machine code. So the actual program that is *executed* is *machine code*. If you look at the machine code listing of any of the programs we have developed so far you will see that some machine

code statements are made up of two bytes: an operation and an address
e.g.



Now consider the problem that the computer has when it reaches line 90 of Program 3.2 during the assembly run. The program has to branch to 'exit' but the line that will define the location of 'exit' has not yet been assembled. That is why we get the message 'No such variable at line 90'.

To overcome this, we have to assemble the program *twice*. On the first run, we switch off the computer's error checking system so that it will ignore problems of the type that it is meeting at line 90. To turn the error checking system off we use the statement OPT 1. **Program 3.3** shows Program 3.2 with the addition of OPT 1. You can see that now that we have added OPT 1 the assembly goes smoothly with no reporting of errors.

```
10 REM BEQ
20
30 PX = &2000
40 [
45 OPT 1
50 .start
60 LDA #0
70 STA &71      \Clear indicator location at start
80 LDA &70      \Put contents of &70 into accumulator
90 BEQ exit     \End run if location &70 contained zero
100 LDA #&FF    \Put the number &FF into accumulator as
                branch indicator
110 STA &71      \Store indicator at memory location &71
120 .exit
130 RTS
140 ]
150 END
```


Test run

```
>RUN
2000
2000          OPT 1
2000          .start
2000 A9 00     LDA #0
2002 85 71     STA &71      \Clear indicator location at start
2004 A5 70     LDA &70      \Put contents of &70 into accumulator
2006 F0 FE     BEQ exit     \End run if location &70 contained
                           zero
2008 A9 FF     LDA #&FF     \Put the number &FF into accumulator
                           as branch indicator
200A 85 71     STA &71      \Store indicator at memory location
                           \&71
200C          .exit
200C 60       RTS
```

Program 3.3

That looks fine (except for the incorrect address for 'BEQ exit' which has assembled as 2006 F0 FE – more about that later) but suppose there is still an error in the program. How are we to know it is there? As things stand in Program 3.3 we would not be able to detect errors so we run the assembly process again with the error reporting turned on. To do this we use OPT 3. To get two runs with different values of OPT we use a FOR ... NEXT loop around the assembly program and use 'OPT pass' where 'pass' is now a variable. If we use a loop

```
FOR pass = 1 TO 3 STEP 2
.
.
.
OPT pass
.
.
.
NEXT pass
```

this will ensure that on the first pass through the loop 'pass' equals 1 in order to turn off the error listing. On the second pass, 'pass' equals 3 which turns the error listing on again. This is called 'two-pass assembly'.

Program 3.4 shows Program 3.3 with the FOR ... NEXT loop inserted to execute the two-pass assembly.

```
10 REM BEQ 2
20
25 FOR pass = 1 TO 3 STEP 2
30   P% = &2000
```

```

40  [
45  OPT pass
50  .start
60  LDA #0
70  STA &71      \Clear indicator location at start
80  LDA &70      \Put contents of &70 into accumulator
90  BEQ exit     \End run if location &70 contained zero
100 LDA #&FF     \Put the number &FF into accumulator as
                branch indicator
110 STA &71      \Store indicator at memory location &71
120 .exit
130 RTS
140 ]
145 NEXT pass
150 END

```

Test run

```

>RUN
2000
2000      OPT pass
2000      .start
2000 A9 00  LDA #0
2002 85 71  STA &71      \Clear indicator location at start
2004 A5 70  LDA &70      \Put contents of &70 into accumulator
2006 F0 FE  BEQ exit     \End run if location &70 contained
                        zero
2008 A9 FF  LDA #&FF     \Put the number &FF into accumulator
                        as branch indicator
200A 85 71  STA &71      \Store indicator at memory location
                        &71
200C      .exit
200C 60    RTS
2000
2000      OPT pass
2000      .start
2000 A9 00  LDA #0
2002 85 71  STA &71      \Clear indicator location at start
2004 A5 70  LDA &70      \Put contents of &70 into accumulator
2006 F0 04  BEQ exit     \End run if location &70 contained
                        zero
2008 A9 FF  LDA #&FF     \Put the number &FF into accumulator
                        as branch indicator
200A 85 71  STA &71      \Store indicator at memory location
                        &71
200C      .exit
200C 60    RTS

```

Program 3.4

If you look at the first pass, you will see that the assembly statement

```
BEQ exit
```

has been assembled into machine code as

```
F0 FE.
```

But in the second listing, 'BEQ exit' has been assembled as

```
F0 04
```

'F0' is machine code for BEQ. On the first pass the assembler couldn't know how far the branch to 'exit' should be, so it inserted a default jump of FE. On the second pass, the assembler had found 'exit' and was able to calculate the correct jump and FE was changed to '04'.

The idea of a relative branch

Let's take a closer look at why 'BEQ exit' has been assembled as F0 04. All assembly language branches are what are called 'relative branches'. That is, the branch instruction says 'branch forward so many bytes' or 'branch backwards so many bytes'. In the instruction

```
F0 04
```

the 04 means 'forward 4 bytes'. It does not, however, mean 'forward from where we are' but 'forward from where the next instruction is'. The next instruction is at &2004. If we go forward a further 4 bytes, we get to &2008 which is 'exit'.

On the face of it, forward and backward branches look the same; but the number of bytes of branch tell you whether a branch is forward or backwards:

Bytes in branch statement	Type of branch
&00 - &7F	Forward
&80 - &FF	Backward

So F0 57 is a forward branch of &57 bytes and F0 C7 is a backward branch of &57 bytes (calculated by the sum &FF - &C7). If that seems a little complicated, don't worry – use labels and let the assembler work it out!

You need to remember that the maximum length of a forward or backward branch is 127 bytes. If you attempt a longer branch than this, you will get an 'Out of range' error message when you try to assemble the program. We will look at this problem in more detail in Unit 4.

SAQ 4

Program 3.4 should branch at line 70 if the contents of the accumulator are equal to 0. Test whether the program works by using suitable numbers in &70.

The other values of OPT

We have used OPT 1 and OPT 3 but two other OPT statements are available: OPT 0 and OPT 2. The effect of all the OPT statements is as follows:

OPT 0	Suppress error messages and suppress the program listing
OPT 1	Suppress error messages but list the program
OPT 2	Report errors but suppress program listing
OPT 3	Report errors and list the program

OPT 0

OPT 0 deserves a special mention since it is important in final versions of programs. Once you have a working, bug free, program, you want to use it but not to see it being assembled on the screen. Thus the finished working assembly subroutine would go into a BASIC program like this:

```
10 ...
20 ...
30 ...           Assembly routine near top of program.
35 ...           Assembled at start of run ready for
                  calling.

40 FOR pass = 1 TO 2
50 [OPT 0         Suppresses listing and error messages
60 ...
70 ...
80 ...
90 ...
100 ]
110 NEXT pass     Program continues here
120 ...           CALL above routine when needed
130 ...
140 ...
```

Another technique is to write:

```
10 ...
20 ...
30 ...           Assembly routine near top of program.
35 opt = 3        Assembled at start of run ready for
                  calling.

40 FOR pass = 0 TO opt STEP opt
50 [OPT opt
60 ...
70 ...
80 ...
90 ...
100 ]
110 NEXT pass     Program continues here
120 ...           CALL above routine when needed
130 ...
140 ...
```

Use the program as shown when developing and testing it. Then change line 35 to `opt = 2` for routine use of the program.

Note that the opening bracket and `OPT` are on the same line in the final version of a program. If you put them on different lines the assembler still outputs the first line of assembly listing to the screen.

3.5 Conditional branches 2

SAQ 5

In section 3.3 we introduced `BNE` and `BEQ`. Programs 3.2, 3.3 and 3.4 used `BEQ`. Rewrite Program 3.4 using `BNE` to perform the same test of the contents of `&70` and test its working with suitable data.

Altogether there are eight conditional branches. We shall now introduce two more.

`BCC` Branch on Carry Clear (i.e. if Carry flag = 0)
`BCS` Branch on Carry Set (i.e. if Carry flag = 1)

These two additional branches test the value of the carry flag which you have already learnt to set and clear. One use of these branch instructions is to check whether the result of an addition exceeded `&FF`. This is how `BCS` is used in Program 3.5. If the answer to the addition is less than `&FF` then the carry flag will still be clear at line 130 after the addition at line 110. In that case, the sum is stored in `&72`. But if the sum exceeds `&FF`, the carry flag will be set and the program will branch at line 130 to the instruction labelled 'large'. This results in the sum (less the lost carry) being stored in `&73`.

```
10 REM BCS
20
30 DIM Z%50
40 FOR pass = 0 TO 3 STEP 3
50   P% = Z%
60   [
70     OPT pass
80
90   .start
100  LDA &70
110  CLC
120  ADC &71
130  BCS large \Will branch if result of addition is > &FF
140  STA &72 \If sum <= &FF then store result in
           location &72
150  RTS \Exits here if sum <= &FF
```

```

160
170 .large
180 STA &73      \Store result of addition (less lost carry)
                  in &73
190
200 RTS          \Exits here if sum >&FF
210 ]
210 NEXT pass
220
240 REM Test routine
250
260 ?&72 = 0
270 ?&73 = 0
280 ?&70 = &18
290 ?&71 = &24
300
310 CALL start
320
330 PRINT "Contents of location &72 = &" ; ~?&72
340 PRINT "Contents of location &73 = &" ; ~?&73

```

Program 3.5

Test run

```

>RUN
Contents of location &72 = &3C
Contents of location &73 = &0

```

Program 3.5

K Load Program 3.5 and run as above. Then repeat with the following data in lines 260 to 290: ?&72 = 0, ?&73 = 0, ?&70 = &82 and ?&71 = &9B. This time you should find that &72 contains 0 and &73 contains &1D.

These results confirm that the branch in line 130 is working correctly. In the first test, the sum is less than &100 so the carry is not set and the sum is stored in &72. In the second test, the correct sum is &11D but this sets the carry flag and leaves &1D in the accumulator. Because the carry flag is set, the program branches at line 100 and stores the result in &73.

SAQ 6

Rewrite Program 3.5 using BCC (Branch on Carry Clear) to branch to 'small' if the sum is less than &100.

3.6 Subroutines

Assembly language makes provision for subroutines in a very similar way to GOSUB in BASIC. The instruction JSR unconditionally transfers program execution to a subroutine.

JSR Jump to Subroutine

The BBC Micro has some built-in subroutines to perform routine tasks which are tedious to program from scratch. Each has a starting location in the Micro's memory at which it can be called. The designers have suggested a name for each routine but the subroutines cannot be called just by their names. We can, however, use the suggested names as labels for the memory locations at which the subroutines start. The first two subroutines that we shall introduce are:

Name	Begins at memory location
OSWRCH	&FFEE
OSNEWL	&FFE7

(The locations &FFEE and &FFE7 are not the actual locations of the subroutines but contain addresses to re-direct the program to the actual locations.)

The names may look odd but they are acronyms for the work they do:

OSWRCH	Operating System Write Character This prints the character corresponding to the number in the accumulator. After printing, the contents of the accumulator are unchanged.
OSNEWL	Operating System NEW Line Produces a new line and carriage return. This leaves &D in the accumulator.

(The equivalent subroutines will be available in other micros but called through different memory locations.)

We will use these subroutines not only to demonstrate the use of JSR but also to enable us to 'see' what is in the accumulator.

You can call a subroutine by an instruction such as JSR &FFEE (this would execute OSWRCH) but programs are much clearer if we use labels wherever we can. So we will label the memory location &FFEE as OSWRCH using BASIC as in line 30 of Program 3.6. Then we can execute the call with JSR OSWRCH.

In Program 3.6 we have used JSR OSWRCH to transfer control of the program to the subroutine called through the memory location &FFEE. This subroutine prints the character in the accumulator (A). It does not print the number in the accumulator but the ASCII character associated with that number. The ASCII codes are listed in Appendix 1.

```

10 REM PRINTING
20
30 oswrch = &FFEE
40 DIM Z%50
50 P% = Z%
60 [
70
80 .start
90 LDA #&2A    \Put '*' into the accumulator
100 JSR oswrch \Print the character in the accumulator
110
120 RTS
130 ]
140
150 REM Test run
160
170 CALL start

```

Program 3.6

Test run

```

>RUN
*>

```

Test runs

With LDA #&2A:

```

>CALL start
*>

```

With LDA #&41:

```

>CALL start
A>

```

In the first run, we loaded the accumulator with &2A (* in ASCII code). In the second run, we loaded it with &41 ('A' in ASCII code). Because each run uses a different version of one line of the program (line 50), we need to re-assemble the program between CALLs.

It's not spectacular, but that is the difference between a high-level language such as BASIC and low-level assembly language. The high-level language will print words, lines, tables, etc. with just a few commands. In assembly language, we have to print one character at a time.

SAQ 7

Write a program to test whether or not location &70 contains 0 or not. If it does, print *. If it doesn't, print !.

3.7 The X and Y registers

Any form of programming involves both decisions and repeated operations. In 6502 assembly language, we use the X and Y registers as counters for repeated operations. These registers do not behave in exactly the same way as the accumulator but two types of accumulator instruction have their equivalents for the X and Y registers.

LDX	LoaD the X register
LDY	LoaD the Y register
STX	STore the X register
STY	STore the Y register

You can put numbers directly into these registers:

```
LDX #&30
LDY #55.
```

Or you can load them from memory:

```
LDX &70
LDY &72.
```

And you can store their contents in memory:

```
STX &71
STY &73.
```

But what is specially useful about the X and Y registers is that you can increase or decrease their contents by 1 by using the statements:

INX	INcrement the X register by 1
INY	INcrement the Y register by 1
DEX	DEcrement the X register by 1
DEY	DEcrement the Y register by 1

There are no corresponding commands to increment or decrement the accumulator.

There are various other instructions which affect the X and Y registers. We shall mention four of the more straightforward ones, even though we don't need them yet.

TAX	Transfer Accumulator to X register
TAY	Transfer Accumulator to Y register
TXA	Transfer X register to Accumulator
TYA	Transfer Y register to Accumulator

'Transfer' is slightly misleading since these four instructions copy the contents of the first register into the second register. The contents of the first register remain unaltered.

This now gives us the facilities we need for counting and, hence, loops.

Program 3.6 printed a *. We shall now use the X register to print five *. The program's flowchart is shown in Figure 3.3.

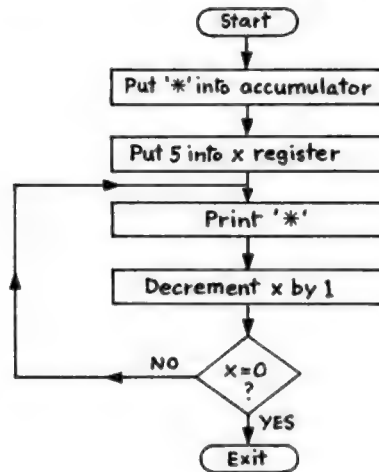


Figure 3.3 Flowchart for printing 5 stars

The program is Program 3.7. This uses a label in a new way: to represent a constant (i.e. a number). In line 30 we have written 'star = ASC "*"'. This makes the label 'star' the same as the number &2A (just as it does when we label memory locations). Then, in line 120, we have written 'LDA #star' which means 'Load the accumulator with the number 'star''. This is a useful device for making programs more readable.

```

10 REM LOOP
20
30 star = ASC "*"
40 oswrch = &FFEE
50 DIM Z%50
60 FOR pass = 0 TO 3 STEP 3
70   P% = Z%
80   [
90     OPT pass
100
110   .start
120   LDA #star \Put '*' into accumulator (stays there
              throughout program)
130   LDX #5   \Put 5 as counter into X
  
```

```

140 .print
150
160 JSR oswrch \Print the '*'
170 DEX      \Decrease counter by 1
180 BNE print \Loop back if more '*'s to print (i.e. if X
           <> 0)

190
200 RTS
210 ]
220 NEXT pass
230
240 REM Test run
250
260 CALL start

```

Program 3.7

Test run

```

>RUN
*****>

```

 Type Program 3.7 and run it. Confirm that the output is *****>. Try adding these lines:

```

45 osnewl = & FFE7
and
185 JSR osnewl.

```

Then re-assemble the program and check that the output is now:

```

>*****
>

```

By using the 'New line and carriage return' subroutine after the stars have been printed, we can bring the cursor onto a new line.

SAQ 8

Write a program to print a column of *s. The number of stars is to be found in the memory location &70 before the program is called.

Program 3.14 (the answer to SAQ 8) is different from Program 3.7 in an important way. Some of the built-in subroutines affect the registers. In Program 3.7, we were using the A and X registers, neither of which is affected by OSWRCH. But in Program 3.15 we were using OSNEWL as well as OSWRCH. OSNEWL does not affect the X register but it does affect the A register, putting &D into it. That is why we loop back to line 160 and re-load the accumulator at line 170. This shows that once you start using built-in routines, you have to remember to consider their effect on the registers. A summary of the main effects appears on pages 456 – 457 of the *User Guide*. For more details, see Chapter 43 of the *User Guide*.

Nested loops

You will be familiar with nested FOR ... NEXT loops in BASIC. For example, Program 3.8 prints

```
*****  
*****  
*****
```

```
10 FOR X = 1 TO 3  
20 FOR Y = 1 TO 5  
30 PRINT "*";  
40 NEXT Y  
50 PRINT  
60 NEXT X
```

Program 3.8

Since we have two counting registers in the 6502, we can nest loops in a very similar way. To produce the same output as Program 3.8, which was in BASIC, we need an assembly language routine such as the one illustrated in Figure 3.4.

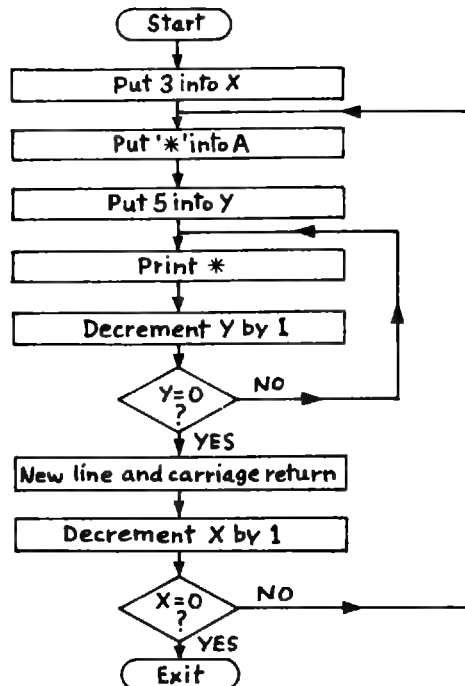


Figure 3.4 Nested loops flowchart

Notice that Y has to be loaded with the number 5 for each of the three rows of stars. X is only loaded once. The program for nested loops is shown in Program 3.9.

```

10 REM NESTED LOOPS
20
30 star = ASC "*"
40 oswrch = &FFEE
50 osnewl = &FFE7
60 DIM Z%50
70 FOR pass = 0 TO 3 STEP 3
80   P% = Z%
90   [
100    OPT pass
110
120    .stars
130    LDX =3          \Set up row counter
140
150    .row
160    LDA #star       \Put '*' into A
170    LDY #5          \Set up column counter
180
190    .column
200    JSR oswrch       \Output '*'
210    DEY             \Decrease column counter by 1
220    BNE column      \Loop back if row not completed
230    JSR osnewl      \New line & carriage return at end of
                      row
240    DEX             \Decrease row counter by 1
250    BNE row         \Loop back if more rows to do (i.e. if
                      X<>0)
260
270    RTS
280  ]
290  NEXT pass
300
310 REM Test run
320
330 CALL stars

```

Program 3.9

3.8 Unconditional jumps

In this unit we have been able to re-direct programs by the use of *branches* and by the use of *subroutines*. In addition to these facilities there is also an unconditional jump in 6502 assembly language

JMP JuMP (unconditional jump)

You can jump to a memory location or to a label. Jumping to a label is the safest procedure since the assembler then works out the correct jump destination, avoiding programming mistakes over addresses. It also makes it easier to re-locate the program into another block of memory if that is needed later. We shall therefore stick to using:

JMP label.

The answer to SAQ 7 (Program 3.13) had two JSR OSWRCH and two RTS statements: at lines 180/190 and at lines 230/250. This is a little untidy so we shall insert an unconditional jump to re-direct the program to the JSR OSWRCH. But for this to work we need a label for the JSR OSWRCH. Let's use the label 'print'. The revised Program 3.13 incorporating JMP is shown in Program 3.10.

```
10 REM BRANCHING
20
30 star = ASC "*"
40 pling = ASC "!"
50 oswrch = &FFEE
60 DIM Z%50
70 FOR pass = 0 TO 3 STEP 3
80   P% = Z%
90   [
100    OPT pass
110
120    .start
130    LDA &70      \Put contents of location &70 into A
140    BNE not
150    LDA #star    \Put '*' into A as zero indicator
160    JMP print
170
180    .not
190    LDA #pling   \Put '!' into A as not-zero indicator
200
210    .print
220    JSR oswrch   \Print not-zero indicator
230
240    RTS
250   ]
260   NEXT pass
270
280 REM Test run
290
300 CALL start
```

Program 3.10

There is another jump instruction:

JMP (xxxx)

which is not the same as JMP xxxx. JMP (xxxx) is described in section 4.6.

3.9 Compare

Three further instructions which affect the flags are the compare instructions. These are:

CMP	CoMPare with accumulator
CPX	ComPare with X register
CPY	ComPare with Y register

Each is followed by a number or a memory location. None of the registers or the contents of any memory location referred to in the instruction is changed by a compare instruction. Only the flags change. We can summarise the flag changes in Table 3.1.

Condition	Effect on flag		
	Z	C	N
Register < value	0	0	1*
Register = value	1	1	0
Register > value	0	1	0*

*ignore when not using two's complement arithmetic

Table 3.1 Effect of 'compare' instructions on the flags

How the flags change

The Z flag. This is the easiest to understand. If there is no difference between the two values the Z flag is set to 1 i.e. $Z = 1$.

The C flag. As long as the result of compare is greater than or equal to zero, the C flag will be set to 1. But if the compare leads to a borrow, then the C flag will be set to 0. So when the value in the register (A, X, or Y) is less than the value compared to, the C flag is set to 0.

The N flag. This flag is set to 1 if the result of a comparison with two's complement signed numbers was negative. We introduced signed numbers in Unit 1 and, for the moment, we are limiting programs to unsigned numbers. We are only mentioning the effect of Compare on the N flag for the sake of completeness.

Testing and branching

A compare instruction would only appear in a program if a branch was to take place depending on the result of the comparison. (Compare instructions are therefore like BASIC's 'IF condition THEN line

number'). We have already met the tests for the C and Z flags and we now introduce the test for the N flag.

BPL	Branch on Plus (i.e. if N = 0)
BMI	Branch on Minus (i.e. if N = 1)

Table 3.1 looks very neat and simple but it reveals a snag. A flag can have the same value under more than one of the conditions \leq and $>$.

SAQ 9

Identify the three possible confusions which can arise from certain flags having the same value under different conditions. Check your answers carefully before going on to SAQ 10.

SAQ 10

Which tests would you use to cause a branch on the following conditions?

Condition	Signed numbers	Unsigned numbers
Register $<$ value		
Register $=$ value		
Register $>$ value		

You should have found that none of the branch instructions work for 'register $>$ value' in either the signed or unsigned case. There seems to be no way of distinguishing this condition from one of the others. The solution is to use two tests in each case. The first test will decide whether or not to apply the second test.

SAQ 11

Which two tests together will cause a branch on 'register $>$ value' for (a) signed numbers and (b) unsigned numbers?

Finally, we might want to branch if:

register \leq value
or
register \geq value.

SAQ 12

Which single test will branch to 'away' on 'register \geq value' for (a) signed numbers and (b) unsigned numbers?

SAQ 13

Which two tests will branch on 'register \leq value' for (a) signed numbers and (b) unsigned numbers? Hint: the successful branch will take place on the first or after both.

Compare summary

Compare is a complicated routine and not one that you would expect to remember in detail. So, when you need to use compare, refer to the following table.

Condition	Test	
	Signed Nos	Unsigned Nos
Register < value	BMI away	BCC away
Register = value	BEQ away	BEQ away
Register > value	BEQ here	BEQ here
	BPL away	BCS away
Register > =value	BPL away	BCS away
Register < =value	BMI away	BCC away
	BEQ away	BEQ away

It is worth noting that when you need to use two tests, it is best to test the more likely event first e.g.

```
BCC
BEQ
```

and not

```
BEQ
BCC
```

Use of BPL and BMI in loops of less than &80

If you have a loop which is to be repeated less than &80 times *and* where you wish to branch out *after* the zero case has been executed, then BPL or BMI is the simplest test to use. For example, the following loop will be repeated six times up to and including the case when X is 0:

```
\ Counting down to zero
LDX # 5
LDA start, x
DEX
BPL loop
```

Assignment C

1. One way of doing multiplication is by repeated addition. Write a program that multiplies a number in location &70 by a number in location &71 by adding the number in location &70 to itself repeatedly. Place the answer in location &72. Run and call your program, testing its operation on suitable data (i.e. positive numbers, the product of which will not exceed the capacity of your program).
2. If the program in (1) is to produce correct answers, the result must not exceed &FF. The multiplier and multiplicand must each not exceed

&0F. Re-write your program so that (a) it checks that neither multiplier nor multiplicand exceeds &0F and (b) if either does exceed &0F, &FF is placed in location &72 and the program exits without performing the multiplication.

Objectives of Unit 3

After studying this Unit you should be able to:

- ☐ State the value of the carry flag after a given series of operations.
- ☐ State the value of the zero flag after a given series of operations.
- ☐ Use BEQ and BNE to branch according to the value of the zero flag.
- ☐ Use OPT and the two-pass assembly process to assemble programs with forward branches.
- ☐ Use BCC and BCS to branch according to the value of the carry flag.
- ☐ Use JSR OSWRCH to print the character in the accumulator.
- ☐ Use JSR OSNEWL to produce a new line and carriage return.
- ☐ Use the X and Y registers to control loops and nested loops.
- ☐ Use JMP to cause an unconditional transfer of program control.
- ☐ Use CMP, CPX and CPY.
- ☐ Test the results of CMP, CPX and CPY using BCC, BCS, BEQ, BNE, BMI and BPL.

Answers to SAQs

SAQ 1

(a) If we did not clear the carry at the start and it happened to be set at 1 then the answer would be 1 too many.

(b) If we cleared the carry between low- and high-byte addition when a carry was required then the carry would fail to take place. This would make the high-byte of the answer 1 short i.e. the number itself would be &100 too small. This shows that you must remember when to use CLC and when not to use it.

SAQ 2

The carry flag values would be:

(a) At line 80, 1. (Set carry to 1 at start.)

At line 90, 0. This is because the *low-byte* subtraction (&53 - &61) requires a borrow from the high-byte subtraction. SBC will only execute a borrow if the carry flag is 0.

(b) At line 80, 1. (Set carry to 1 at start.)

At line 90, 1. Here the low-byte subtraction (&A2 - &77) does not require a borrow from the high-byte subtraction so the carry flag is left set at 1 to prevent the borrow.

SAQ 3

- (a) Result is 0 so $Z = 1$.
- (b) Result is not 0 so $Z = 0$.
- (c) Result is 0 so $Z = 1$.

SAQ 4

First see what happens when the content of &70 is not 0.

```
>?&71 = 0           Set &71 to 0 to start.
>?&70 = 45
>CALL start
>P.?&71
      FF
```

We started with the contents of &71 equal to 0 and ended with its contents &FF (= &2A). The program *did* branch.

Next try with the contents of &70 equal to zero.

```
>?&71 = 0           Re-set &71 to 0.
>?&70 = 0
>CALL start
>P.?&71
      0
```

At the end of this run, &71 is still 0, so the program did *not* branch.

SAQ 5

```
10 REM BRANCHING
20
30 DIM Z%50
40 FOR pass = 0 TO 3 STEP 3
50   PX = Z%
60   [
70     OPT pass
80
90     .start
100    LDA &70      \Put contents of &70 into accumulator
110    BNE there    \If not zero then branch
120    RTS          \Exits here if contents of &70 are zero
140    .there
150    LDA 9&FF    \Put &FF into accumulator as indicator
160    STA &71      \Store indicator at location &71
170
180    RTS          \Program exits here if branch happened
190   ]
200 NEXT pass
210
220 REM Test data
230 ?&70 = 34
240 ?&71 = 0
250
260 CALL start
270
280 PRINT "Contents of location &71 = &" ; "?&71"
```

Test run

```
>RUN
Contents of location &71 = &FF      The program did branch.
```

Program 3.11

[K] Change the test data to ?&70 = 0 and ?&71 = 0 and then run Program 3.11 again. This time the contents of &71 after the run should be zero, showing that the program did not branch.

SAQ 6

```
10 REM BRANCHING
20
30 DIM Z%50
40
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80     OPT pass
90
100    .start
110    LDA &70
120    CLC
130    ADC &71
140    BCC small    \Will branch if sum <=&FF
150    STA &72      \If sum >&FF the sum (less lost carry)
                   stored at &72
160    RTS          \Exits here if sum >&FF
170
180    .small
190    STA &73      \Store sum here if <=&FF
200
210    RTS          \Exits here if sum <=&FF
220  ]
230  NEXT pass
240
250 REM Test data
260 ?&72 = 0
270 ?&73 = 0
280 ?&70 = &22
290 ?&71 = &2D
300
310 CALL start
320
330 PRINT "Contents of Location &72 = &" ; "?&72
340 PRINT "Contents of Location &73 = &" ; "?&73"
```

Program 3.12

Test run

```
>RUN
Contents of location &72 = &0           The program branched.
Contents of location &73 = &4f
```

K Run Program 3.12 as above. Then change the test data in lines 260 to 290 to ?&70 = &86, ?&71 = &91, ?&72 = 0 and ?&73 = 0 and run the program again. Check that the contents of &72 and &73 are &17 and 0, this time showing that the program did not branch.

SAQ 7

```
10 REM STAR OR PLING
20
30 star = ASC "*"
40 pling = ASC "!"
50 oswrch = &FFEE
60
70 DIM Z%50
80
90 FOR pass = 0 TO 3 STEP 3
100  P% = Z%
110  [
120    OPT pass
130
140    .start
150    LDA &70    \Put contents of location &70 into A
160    BNE not    \Branch if contents of A not zero
170    LDA #star  \Put '*' into A as zero indicator
180    JSR oswrch \Print zero indicator
190    RTS
200
210    .not
220    LDA #pling \Put '!' into A as not-zero indicator
230    JSR oswrch \Print not-zero indicator
240
250    RTS
260  ]
270  NEXT pass
280
290
300 REM First test
310 ?&70 = 0
320 PRINT "Action when contents of &70 = 0:"
330 CALL start
340
350 REM Second test
```

```

360 ?&70 = 27
370 PRINT ' "Action when contents of &70 = 27:"
380 CALL start

```

Program 3.13

Test run

```

>RUN
Action when contents of &70 = 0:
*
Action when contents of &70 = 27:
!>

```

SAQ 8

The key points here are:

(a) that you need to use LDX &70 to put the number in &70 into the X register
and

(b) that OSNEWL will need to be called after each star is printed.

```

10 REM STAR COLUMN
20
30 oswrch = &FFEE
40 osnewl = &FFE7
50 star = ASC "*"
60 DIM Z%50
70
80 FOR pass = 0 TO 3 STEP 3
90   P% = Z%
100  [
110    OPT pass
120
130    .start
140    LDX &70      \Get counter from location &70
150
160    .print
170    LDA #star    \Put '*' into A
180    JSR oswrch    \Print '*'
190    JSR osnewl    \Carriage return and new line
200    DEX          \Decrease counter
210    BNE print     \Loop back if more '*'s to do (i.e. if X<>0)
220
230    RTS
240  ]
250  NEXT pass
260
270 REM Test run
280 INPUT "Enter number of stars required " number
290 ?&70 = number
300
310 CALL start

```

Program 3.14

Test run

```
>RUN
Enter number of stars required 5
*
*
*
*
*
>
```

Program 3.14

SAQ 9

C = 1 when 'register = value' and when 'register > value'.

Z = 0 when 'register < value' and when 'register > value'.

Signed numbers only:

N = 0 when 'register = value' and when 'register > value'.

SAQ 10

Condition	Signed Nos	Unsigned Nos
Register < value	BMI away	BCC away (1)
Register = value	BEQ away (3)	BEQ away (2)
Register > value	(4)	(4)

(1) If you said 'BNE away' then the branch would also take place on 'register > value'.

(2) If you said 'BCS away' then the branch would also take place on 'register = value'.

(3) If you said 'BPL away' then the branch would also take place on 'register > value'.

(4) There is no single instruction that will do the job. We have to use two instructions in each case. The method is described in the next section of the Unit.

SAQ 11

(a) The problem is to distinguish combinations of values of N and Z. To do this we use BEQ followed by BPL but only BPL branches to 'away'. BEQ branches to 'here', i.e. it doesn't really branch at all because the label 'here' will follow the test:

```
BEQ here
BPL away
.here
etc.
```

This is illustrated in Figure 3.5.

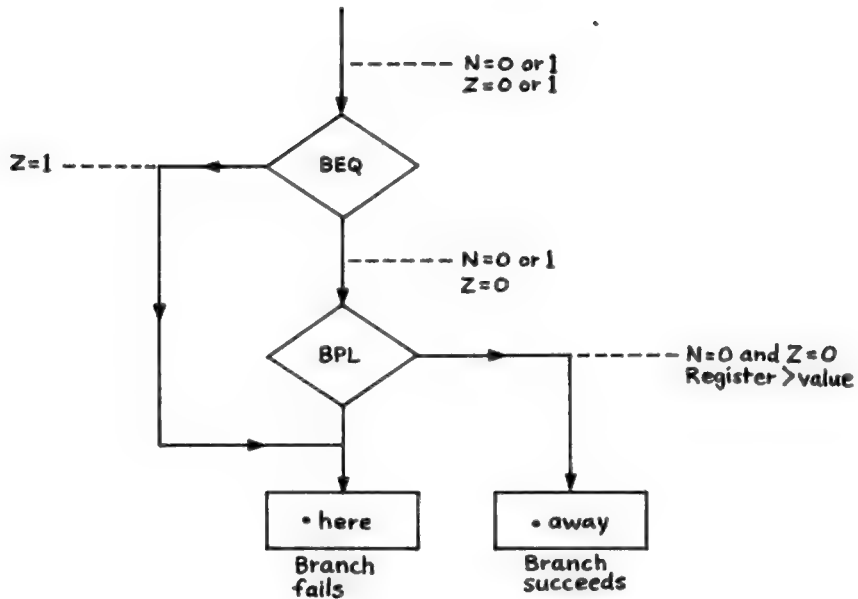


Figure 3.5 Test for 'register > value' (signed numbers)

(b) Here the problem is to distinguish between combinations of values of Z and C. To do this we use 'BEQ here' and 'BCS away':

```

BEQ here
BCS away
.here
etc.

```

This is illustrated in Figure 3.6.

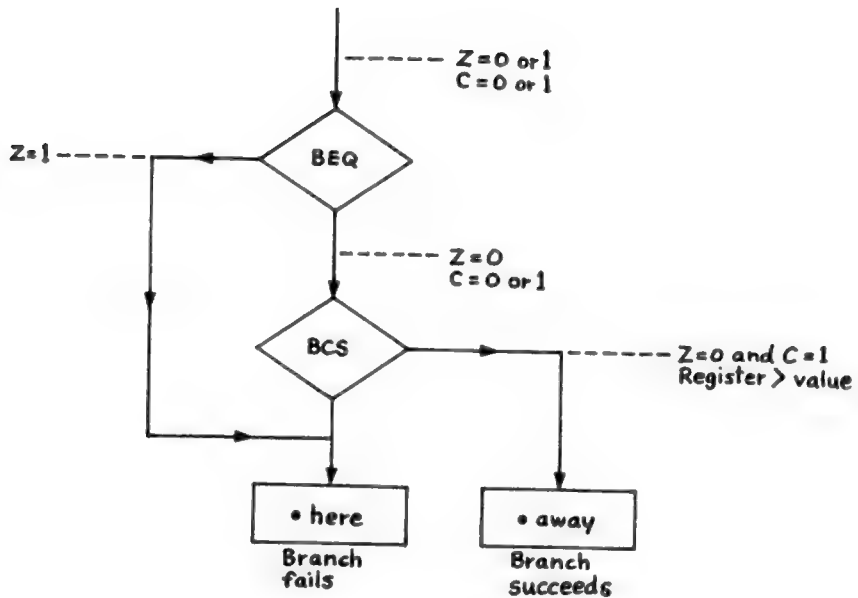


Figure 3.6 Test for 'register > value' (unsigned numbers)

SAQ 12

(a) BPL away (b) BCS away

SAQ 13

- | | |
|--------------|--------------------|
| (a) BMI away | (Will branch if <) |
| BEQ away | (Will branch if =) |
| (b) BCC away | (Will branch if <) |
| BEQ away | (Will branch if =) |

Note: In both cases, both tests branch to 'away'.

UNIT 4

Addressing modes

Introductory note

- 4.1 What is addressing?
- 4.2 Immediate mode
- 4.3 Zero Page mode
- 4.4 Absolute mode
- 4.5 Implied mode
- 4.6 Indirect addressing
- 4.7 Absolute indexed mode
- 4.8 Zero Page indexed mode
- 4.9 Relative mode
- 4.10 Accumulator mode
- 4.11 Indexed indirect mode
- 4.12 Indirect indexed mode

Assignment D

Objectives of Unit 4

Answers to SAQs

Introductory note

This unit is unavoidably different from the rest of the course in that it is to some extent a reference unit. It contains some activities and self-assessment work, but less than in other units.

You use addressing modes every time you write an assembly language statement. You have already learnt some of the modes. But since there are many modes, some of which are rather difficult, there is no obvious point in the course at which to introduce them all. However, to appreciate the modes, you need at some stage to see them all together. So it makes sense to introduce them all now.

Don't be put off if you find this Unit a little difficult at times and too much to remember all at once. Treat its sections at the following levels of importance.

4.1 What is addressing?	Very important
4.2 The immediate mode	You have used these already.
4.3 The zero page mode	{ This Unit puts them into perspective amongst the range of address modes
4.4 The absolute mode	
4.5 The implied mode	
4.6 Indirect mode	Not used much in this book
4.7 Absolute indexed mode	Very important
4.8 Zero page indexed mode X	A variation of 4.7
Zero page indexed mode Y	Not used in this book
4.9 Relative mode	You have already used this
4.10 Accumulator mode	Used in the next Unit
4.11 Indexed indirect	Skip this if difficult
4.12 Indirect indexed	Very important

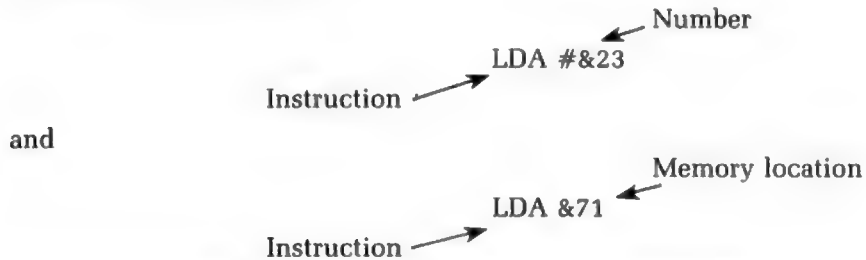
This multiplicity of modes contributes to the versatility of the 6502 whilst at the same time making it look a little daunting. So, at a first run through the book, you might like to

Study carefully	4.1, 4.7 and 4.12
Read through	4.2 to 4.5, 4.9, 4.10 and 4.11

and leave the other sections until you need them to solve a particular problem.

4.1 What is addressing?

Most assembly language instructions are in two parts: an assembly instruction code followed by a number or a memory location (called the 'operand'). You have met

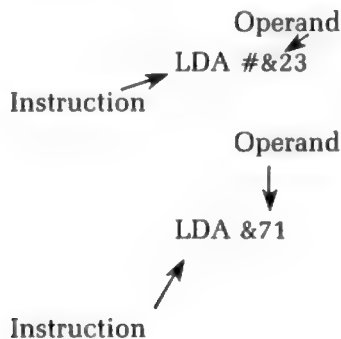


If you look at these two instructions when they are assembled, you will find that:

LDA #&23
becomes
A9 23

LDA &71
becomes
A5 70

So LDA does not assemble as one fixed code but varies according to its operand. If a number is loaded directly (not from memory), then LDA assembles as A9. If the accumulator is to be loaded from zero page, then LDA assembles as A5.



These two examples show two different addressing modes: the 'immediate mode' and the 'zero page mode'.

4.2 Immediate mode

In immediate mode, the operand is an 8-bit number preceded by # which is read as 'immediate' e.g.

LDA #&23	Load the accumulator immediate &23
CMP #0	Compare to immediate 0

You have been using this mode since Unit 2. What we haven't said is which instructions can use the immediate mode. It can, in fact, be used after 11 of the 6502 instructions. These are : ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA and SBC.

(In some assemblers you'll see 'Load Accumulator Immediate' written as LDAIM.)

4.3 Zero Page mode

This again has occurred many times in this course. In the Zero Page mode, the instruction is followed by the address of a memory location in Zero Page i.e. a number between 0 and &FF. Examples of this are:

```
LDA &72
STA &73
```

21 instructions can use this mode: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX and STY.

4.4 Absolute mode

This is the same as the Zero Page mode except that a memory location outside Zero Page is being addressed. This immediately brings about an important difference: any Zero Page address is 1 byte long whereas any other address in memory will be 2 bytes long. **Because of this, any instruction which addresses Zero Page involves fewer numbers than the same instruction addressing a memory location elsewhere. A Zero Page instruction is executed faster than one which addresses a location elsewhere.** Examples of the absolute mode are:

```
LDA &3000
LDY &2348
```

23 instructions can use this mode: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX and STY.

4.5 Implied mode

In the implied mode, nothing follows the instruction since the instruction needs no further address. The address is *implied* in the instruction itself. For example:

CLC needs no further address
and

INY needs no further address

In both cases the instructions have implicit addresses.

You will recognise this as a mode which you have already used. It can be used with 25 different instructions: BRK, CLC, CLD, CLI, CLV, DEY, DEX, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS and TYA.

Any instruction which uses the implied mode does not use any other mode.

4.6 Indirect addressing

This is used by one instruction only: JMP. It consists of the format

JMP (&xxxx)

where &xxxx is an address in memory. (Note carefully the position of the brackets). At &xxxx another address is stored to which the program is to branch. This action is illustrated in Figure 4.1.

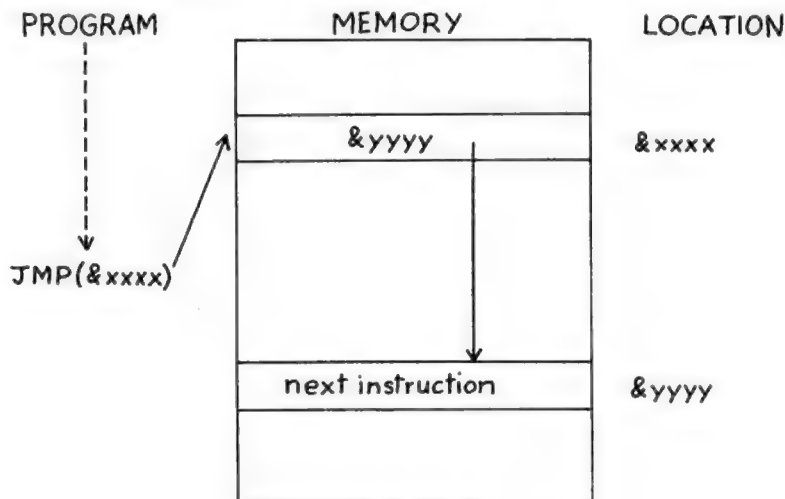


Figure 4.1 Action of `JMP (&xxxx)`

At JMP (&xxxx), the program loads the program counter from memory location &xxxx at which is stored the memory location number &yyyy. The program then goes to &yyyy where the next instruction will be found. This may look like a long way round to get to &yyyy but it can make sense when you need to vary the final destination of the jump but cannot vary the address in the jump instruction. Instead, you arrange to vary the contents of &xxxx itself. This is a more advanced technique which we shall not need to use in this book.

You may find it helpful to compare indirect addressing with BASIC when you write:

```
A = 100 : GOTO A
```

4.7 Absolute indexed mode

This mode has two versions:

Absolute indexed X Absolute indexed Y
--

Its purpose is to allow the address to which the program goes to be modified by the value in the X or Y register. For example,

```
LDA &70, X
```

will load the accumulator with the contents of memory location $\&70 + X$. So if $X = 5$,

```
LDA &70, X
```

is effectively

```
LDA &75.
```

The value of this mode is that it allows us to work systematically through a list of data stored in sequential memory locations.

Example 1

Fifty 1-byte numbers are stored in memory starting at &3000. (Memory at &3000 is free for your use in Mode 7). Search the list to find out whether the number &39 is in the list. If it is, store the number &FF in &70. Otherwise store 0 there.

The program is shown in Program 4.1. It incorporates two new ideas. First, we have used a label for the memory location &70. At line 40 we have put

```
flag = &70
```

This then allows us to put 'flag' in the program in place of &70 so making the program much more readable. From now on we will increasingly use labels for memory locations. Second, we have used a useful device for setting the value that 'flag' will hold. It starts at zero

(line 120). If we don't find &39, the flag stays untouched. But if we do find &39, we decrement the flag by 1. Note that 0 - 1 produces &FF, since we are dealing with 8-bit numbers. If the flag is decremented at line 240, it goes from 0 to &FF.

```

10 REM LIST SEARCH
20
30 DIM Z% 50
40 flag = &70
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80     OPT pass
90
100    .search
110    LDA #0
120    STA flag      \Initialise flag
130    LDX #&32      \Initialise counter
140
150    .loop
160    LDA &3000,X \Load A with next item
170    CMP #&39     \Compare to search value
180    BEQ found     \Branch if found
190    DEX
200    BNE loop      \Loop back if more items
210    RTS           \Not found
220
230    .found
240    DEC flag      \Set flag to 'found' (&FF)
250    RTS
260  ]
270  NEXT pass
280
290 REM Test routine
300 REM Put random numbers into the memory block
310 PRINT "List of numbers put into memory block." '
320 FOR I = &3000 TO &3032
330   ?I = RND(&FF)
340   PRINT ; "?I ; " " ;
350 NEXT
360
370 REM Call search routine
380 CALL search
390
400 PRINT " " "Result of search:"
410 IF ?&70 = 0 THEN PRINT "&39 was not in list." ELSE PRINT
    "&39 was in list."

```

Program 4.1

The critical parts of this program are:

```
130 LDX #&32
140
150 .loop
Loops back 160 LDA &3000,X
if X > 0   ...
           ...
           ...
190 DEX
200 BNE loop
```

We have chosen to check the list from the top down. At the first pass, $X = \&32$ so line 160 is effectively

160 LDA &3032

and the number at &3032 is loaded into the accumulator. Then at line 190, X is decremented by 1. On the second pass, line 160 is effectively

LDA &3000 + &31.

i.e. the accumulator is loaded with the number in &3031. The passes go on in this manner, successively loading the accumulator with the contents of &3030, &302F, &302E etc. until the last location of &3000 is reached. This occurs when $X = 0$ which is detected by line 200. Checking the list backwards is more efficient in this case (and in many other cases) since it avoids having an extra `CPX #&32`.

Test runs

```
>RUN
List of numbers put into memory block.

C1 C0 EC 31 39 AF FF 1 B0 E4 B0 3C 63 81 8F FF 18 FF 84 80 B1 F
E0 80 80 33 31 B0 51 8C 50 88 9D 41 31 14 29 C8 2B 7F B1 BF 5 CF
BC 51 B1 40 AF 94 B1
```

Result of search: &39 was in list.

```
>RUN
List of numbers put into memory block.

D 1D F1 40 F3 58 48 19 50 2 EF 89 81 35 CD F1 10 BD BC 77 28 64
41 81 F7 99 FB 48 FE 3F 41 1D 31 38 D0 1 31 4F EB 31 40 2D 42 70
80 18 CF 85 10 32 FC
```

Result of search: &39 was not in list.

☒ Type and test Program 4.1.

SAQ 1

What is the maximum length of a list that could be searched in this way?

SAQ 2

Write a program using the absolute indexed mode to transfer 50 1-byte numbers stored sequentially in memory starting at &3001 to locations &3801 to &3832.

Note:

(a) We have worked these examples using the X register. We could equally well have used the Y register. The 15 instructions with which you can use absolute indexed addressing are: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC and STY.

(b) In Program 4.1, we could just as well have written

```
LDA #&39
CMP &3000, X
```

4.8 Zero Page indexed mode

Zero Page indexed X

This is the same as absolute indexed X except that the address being modified by X is in Zero Page e.g.

```
LDA &70, X
```

SAQ 3

Is there any practical difference between absolute indexed X and Zero Page indexed X? Hint: think of the possible range of values of X.

Zero Page wrap-round

SAQ 3 demonstrates a condition called 'Zero Page wrap-round'. When an attempt is made to use a Zero Page mode with an address that would go above Zero Page, the 6502 ignores the msb of the address e.g. `LDA &70, X` with `X = &A0` gives an address above Zero Page. That address is treated as `&10` which is in Zero Page.

Zero page indexed X mode can be used with 16 different instructions: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA and STY.

Zero Page indexed Y

This is not similar to Zero Page indexed X since it can only be used after the instructions LDX and STX. This mode allows us to use the Y register to modify the locations in Zero Page at which the X register loads. This mode is not often used.

4.9 Relative mode

This is just a new name for a mode you have already learnt to use. It is the mode used by the branch instructions (and only by the branch instructions). In the relative mode, the program is told to branch to an address relative to its current address. For example, a program may be told to branch 35 bytes forward or 27 bytes backwards. This is tricky to work out. If you stick to branching to labels, the assembler works out the number of bytes in the jump for you. We shall not therefore bother with the forward and backward branching tables needed when you have to work out the length of jump for yourself.

However, you do need to bear one point in mind about the relative mode. The maximum *forward* or *backward* jump is 127 bytes. If you try a longer jump then you will get an error message when you try to assemble the program.

SAQ 4

Longer jumps are possible but not in relative mode. How could you synthesise a relative mode branch instruction which would jump *more than 127 bytes*? Hint: follow it with which instruction?

4.10 Accumulator mode

It applies only to ASL, LSR, ROL and ROR. Thus you may write

```
ASL A
LSR A
ROL A
ROR A
```

No other instruction may address the accumulator. We use it in the next unit.

4.11 Indexed indirect mode

This mode combines indirect addressing and indexed addressing. It has a limited purpose which is best explained by an example.

Example 2

A program will obtain data from one of 5 addresses according to the value of the variable X which occurs in the algorithm. You don't know what these addresses are at the time of writing the program but you do know that they will be stored in &80 and &81, &82 and &83, &84 and &85, &86 and &87, and &88 and &89.

The new address format that we need to solve this problem is

```
... (&bb, X)
```

where &bb is a base address in zero page and X is the value in the X register. (Note carefully the position of the brackets). Examples of this address format are:

```
LDA (&80, 2)
STA (&80, 3)
```

So, returning to the example, if the program included the statement

```
LDA (&80, X)
```

and X had the value 4 then the effect would be:

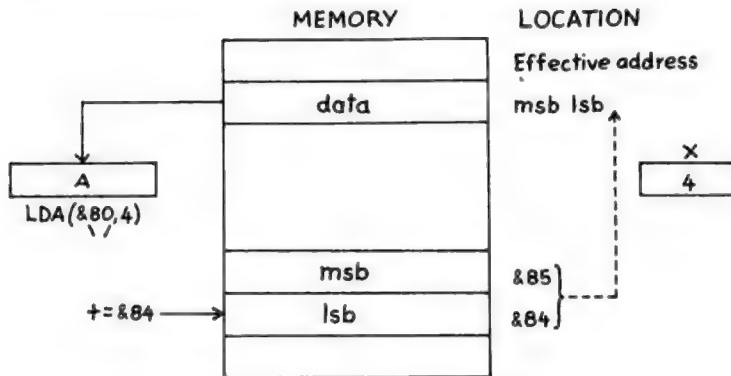


Figure 4.2 Effect of `LDA (&80, 4)`

The steps in Figure 4.2 are:

- ☐ `LDA (&80, 4)`.
- ☐ Base address added to index: $\&80 + \&4 = \&84$.
- ☐ CPU addresses $\&84$ and $\&85$ i.e. two bytes are addressed starting at $\&84$.
- ☐ At $\&84$ there is the low-byte of the effective address (lsb).
- ☐ At $\&85$ there is the high-byte of the effective address (msb).
- ☐ CPU addresses msb lsb.
- ☐ Data from msb lsb is loaded into A.

The base address must be in Zero Page, as must be all the indexed addresses.

X must increment by 2 to step through the indexed addresses since each indexed indirect mode address occupies two bytes of Zero Page.

This mode is rarely used since it takes up valuable Zero Page space which you would not normally be able to spare.

4.12 Indirect indexed mode

In this mode the computer loads an address at which a base address is stored. It then goes to the base address from which indexing takes place. The first address must be in Zero Page. The format of such an address is:

.... (&aa), Y

where &aa is a Zero Page address and Y is the value in the Y register. Note that the brackets are in a different position from the indexed indirect X mode.

Examples of this mode are:

```
LDA (&70), Y
STA (&75), Y
```

The action of LDA (&70), Y (where $Y = 3$) is explained in Figure 4.3.

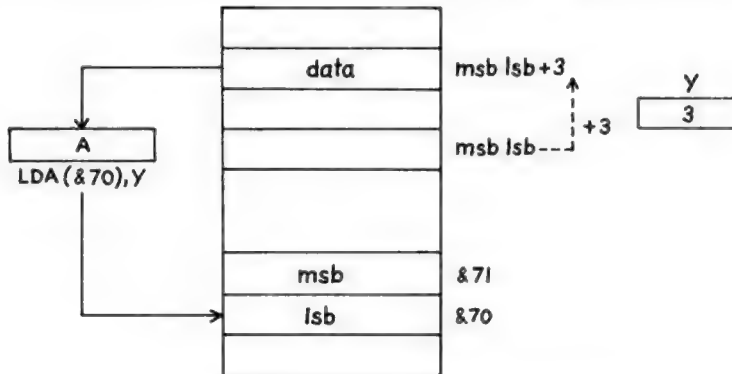


Figure 4.3 How LDA (&70), Y loads from msb lsb + 3

The steps are as follows:

- ☐ LDA (&70), Y loads from an address held in locations &70 and &71.
- ☐ These store the low- and high-byte of the base address: msb and lsb.
- ☐ The CPU takes the base address and adds the index in Y (3) to give msb lsb + 3.
- ☐ This is the effective address from which the accumulator will be loaded.

As you will see from the rest of this book, this mode is an extremely useful one and is used again and again in a very wide range of problems.

Example 3

Write a program to clear (set to 0) &100 memory locations starting from a base address which is stored at &80 (low-byte) and &81 (high-byte).

Solution

Note first that we can't use the absolute indexed mode to solve this problem since that requires us to know the base address of the list. We are not told the base address in this example and know only that it will be stored in &80 and &81. Hence we must use indirect indexed mode.

The program is shown in Program 4.2. Notice the form of the REM statement in line 30. This reminds us that 'base' will be storing a 2-byte number. We have used one additional new technique in this program where, in line 260, we have written 'base?1'. This refers to the address 1-byte above 'base' and, in line 260, pokes &30 into that address.

```

10 REM PAGE WIPE
20
30 base = &80 : REM Plus &81
40 DIM Z%50
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80     OPT pass
90
100    .wipe
110    LDY #0          \Initialise byte counter
120    LDA #0          \Initialise A
130
140    .next
150    DEY
160    STA (base),Y    \Clear a location
170    BNE next        \Loop back if more to clear
180
190    RTS
200  ]
210  NEXT pass
220
230 REM Test routine
240 REM Set base address
250 ?base = 0
260 base?1 = &30
270
280 REM Fill page at base with random numbers
290 PRINT "Test data put into memory:" '
300 FOR I = &3000 TO &30FF
310   ?I = RND(&FF)
320   PRINT ; "?I ; " " ;
330   NEXT
340
350 REM CALL clear routine
360 CALL wipe
370
380 REM Check the result of same memory locations after
    calling 'wipe' routine:" '
400 FOR I = &3000 TO &30FF
410   PRINT ; "?I ; " " ;
420   NEXT
430 PRINT

```

Program 4.2

Test run

```
>RUN
```

Test data put into memory:

1B F4 C9 7D 67 68 31 F7 39 E9 EB D4 81 62 45 B0 1 67 10 7F 65 64 CE 1F
74 EB C4 69 A5 63 2C 45 46 7B 80 7A 7D 68 FE 94 5D 27 F6 AD B8 A3 2E
16 63 4B C4 3C 20 D4 B7 64 75 8C 3E FA 7A 45 DB EA 2F D9 9E 6D 1E 4B
2C C5 8C 89 90 95 D4 E7 9F D7 EC 28 9D B0 16 C6 4A DC 7E 3F AD F8 77
71 C4 50 31 CA 47 4B 80 3A D4 BA D1 4B C1 4C E9 EA 47 68 A1 9E F3 DF 9
F DC 81 4 5C 45 E4 24 29 3D B0 9F A7 DB 8F 21 EB 1E CA 6 1D D CA C2 11
AC F 65 50 90 2E 6D 79 46 53 A2 EB 46 8F E1 3F 25 F6 73 90 C D4 EA 6F
76 31 A7 7D 59 5E 6C 24 56 23 EF 5C 6B 6B 60 E3 74 92 B9 CF 9E 76 29
9C 66 7B 9D 59 D9 2A 3C B2 6 1E 7D 46 F C8 E6 A4 B4 21 B 74 3E 12 F6 8
3B 5B AA A CA 83 1A 6E 93 AF 9C 3A EC F9 24 48 8C 15 EB 68 7F 55 63 F
D 6F 37 3A 68 E1 93 8E CF AC 9C 37 9E E2 85 DE 68 72 13 C6

Contents of same memory locations after calling 'wipe' routine:

[illegible]

K Type and run Program 4.2. Check its working. Then try it with a block of memory at &2000 upwards.

SAQ 5

In SAQ 2 we used the absolute indexed mode to move a block of data. Write a program using indirect indexed addressing to move 60 items of data from the block of memory starting at &3000 to the block starting at &3800.

The solution which we have given in the answer section works provided that (a) no more than one page is to be moved and (b) the position of the new block does not overlap the position of the old block. Let's first look at how to deal with overlapping blocks in the single page move routine.

Example 4

The program has to allow for two possibilities. Either the new block is above the old block (Figure 4.4a) or it is below it (Figure 4.4b).

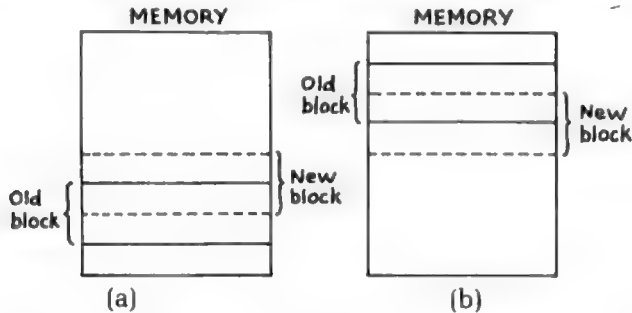


Figure 4.4 The new block is either above or below the old block

In case (a), we must move the old block to the new by starting at the top of the old block and moving down through it. In case (b), we must move the other way – start at the bottom of the old block and work up through it. The program has therefore to have two routines. If 'old block < new block' then we use the 'down' routine. If 'old block > new block' then we use the 'up' routine.

The program is shown in Program 4.3. Because we are moving data around memory, we want the machine code well out of the way. We have therefore lowered HIMEM by &100 bytes and put the machine code in the space created.

```

10 REM PAGE MOVE WITH OVERLAP
20
30 MODE 3
40 HIMEM = HIMEM - &100
50 oldlow = &70
60 oldhigh = &71
70 newlow = &72
80 newhigh = &73
100
110 FOR pass = 0 TO 3 STEP 3
120   P% = HIMEM
130   [
140     OPT pass
150
160     .move_with_overlap
170     LDA oldlow          \Subtract addresses
180     SEC                  \of the
190     SBC newlow          \two blocks
200     LDA oldhigh        \to find whether
210     SBC newhigh        \move is up or down
220     BCS down           \Move down if new page is below old page
230
240     .up                 \Else move up
250     LDY #&FF
260     .nextup
270     LDA (oldlow),Y      \Get byte from top of old block
280     STA (newlow),Y      \Store it at top of new block
290     DEY                 \Decrement byte counter
300     CPY #&FF           \Was last item done the zero'th item?
310     BNE nextup         \If not, loop back for next item
320     RTS                \Else exit with move finished
330
340     .down
350     LDY #0              \Set byte counter to 0
360     .nextdown
370     LDA (oldlow),Y      \Get byte from bottom of old block
380     STA (newlow),Y      \Store it at bottom of new block
390     INY                 \Increment byte pointer

```



```

400   BNE nextdown      \Loop back if page not finished
410   RTS               \Else exit with page moved
420   J
430   NEXT pass
440
450 PRINT ' "First test  to move data from base at &3000 to
      base at &3020:"
460 REM Uses 'up' option
470 ?oldlow = &0
480 ?oldhigh = &30
490 ?newlow = &20
500 ?newhigh = &30
520 PROCtest
530
540 PRINT ' "Press space bar for next test."
550 REPEAT : UNTIL GET = 32
560
570 PRINT ' "Second test to move data from base at &3020 to
      base at &3020:"
580 REM Uses 'down' option
590 ?oldlow = &20
600 ?oldhigh = &30
610 ?newlow = &0
620 ?newhigh = &30
630 PROCtest
640
660 END
670 DEF PROCtest
675 @% = 4
680 PRINT'
690 start = ?&71 * &100 + ?&70
700 FOR I = start TO start + &FF - 1
710   ?I = RND(&FF)
720   PRINT ~?I, ;
730   NEXT
740
750 CALL move_with_overlap
760
770 start = ?&73 * &100 + ?&72
780 PRINT'
790 FOR I = start TO start + &FF - 1
800   PRINT ~?I, ;
810   NEXT
815 @% = 10
820 ENDPROC

```

Program 4.3

Test run

>RUN

First test to move data from base at \$3000 to base at \$3020:

```
81 34 BD F7 56 58 6F 51 F7 CA AC D7 7A D 4F 1 A 6C A9 5E
7A 19 29 74 B6 13 9C 7B BC 44 E7 63 B5 8D 7D BC FE 3E FE 31
2A EF E3 DF 63 68 9D 34 BC 4B 8C CA EB F4 64 95 8E 89 EC AD
52 54 A1 38 2 DD 1E 49 1 B4 B1 35 DB DD 8B 76 53 EA 84 3D
66 F0 EB B4 CB 36 E9 2D 95 3A 3A D9 78 8F D8 D7 6B EB 5A 22
2F EF 5F 6 FF ED 35 A8 E2 6C D3 5D 4B E9 38 3D DF 21 70 C8
3 7B 67 3D 27 9B BC 7F 36 2 39 6 6B C7 D8 64 DB DF 83 FC
A6 9A E EF 7C D1 53 48 A DF 19 6B A3 C5 A4 6 E7 F8 19 48
DA AC D D7 7F BB 95 91 6B 5 C7 E8 5F 61 3B 8 8 84 CA 16
93 A6 D3 CF 15 1A 51 34 6F E4 40 F0 19 BA 96 5 DF F0 73 81
3 CF D8 9 FC 48 31 A 67 EB BC 3A D 29 34 D7 11 E9 6B D0
32 A0 34 38 D 5E BB 9F 63 35 41 4B 8C 2B FC 1 19 16 2D E0
39 C7 E0 8B 36 53 A2 AC B4 25 A8 21 5D 1C 5D
```

```
81 34 BD F7 56 58 6F 51 F7 CA AC D7 7A D 4F 1 A 6C A9 5E
7A 19 29 74 B6 13 9C 7B BC 44 E7 63 B5 8D 7D BC FE 3E FE 31
2A EF E3 DF 63 68 9D 34 BC 4B 8C CA EB F4 64 95 8E 89 EC AD
52 54 A1 38 2 DD 1E 49 1 B4 B1 35 DB DD 8B 76 53 EA 84 3D
66 F0 EB B4 CB 36 E9 2D 95 3A 3A D9 78 8F D8 D7 6B EB 5A 22
2F EF 5F 6 FF ED 35 A8 E2 6C D3 5D 4B E9 38 3D DF 21 70 C8
3 7B 67 3D 27 9B BC 7F 36 2 39 6 6B C7 D8 64 DB DF 83 FC
A6 9A E EF 7C D1 53 48 A DF 19 6B A3 C5 A4 6 E7 F8 19 48
DA AC D D7 7F BB 95 91 6B 5 C7 E8 5F 61 3B 8 8 84 CA 16
93 A6 D3 CF 15 1A 51 34 6F E4 40 F0 19 BA 96 5 DF F0 73 81
3 CF D8 9 FC 48 31 A 67 EB BC 3A D 29 34 D7 11 E9 6B D0
32 A0 34 38 D 5E BB 9F 63 35 41 4B 8C 2B FC 1 19 16 2D E0
39 C7 E0 8B 36 53 A2 AC B4 25 A8 21 5D 1C 5D
```

Press space bar for next test.

Second test to move data from base at \$3020 to base at \$3020:

```
35 2F 1A 72 6B A7 A4 EB CA 7F DE 89 51 B4 3D BD D BD 24 9C
EB A4 2D 21 C8 C 11 CF 63 5F 82 93 D 80 14 A4 7F D6 85 40
1 6E 53 FE 41 5C A5 F3 E0 53 3F B0 97 6B 87 89 5F 1F 8F 30
CF D D 7C D3 57 25 E0 7C 2D E8 35 A9 6C D9 DA A1 5F D4 8B
3D D3 5E 43 64 C0 50 57 F3 72 F6 BB 87 A4 D6 D6 DD 8E 84 B0
DC A8 73 AA 58 69 25 8E 2A D3 A9 70 C 99 7 88 9E EE 44 E0
8C 5C 1 5D ED EF 81 5C FF F 26 EC A4 45 AE FA 35 BA 2C C4
6 72 D2 F4 83 E3 19 5D 78 90 CB B0 44 D0 F5 FA 68 55 C5 44
5E 8E DC D2 70 41 5C F3 8D A4 73 85 87 CA 88 29 9C D6 28 5C
4 FA 4C 1B C1 4 94 7 77 CC 28 40 C3 94 B F4 4D B9 67 1
F0 77 E8 68 44 85 5A 81 5F 51 F2 67 9F 97 F0 B8 58 7A 65 94
F6 1 24 50 5 48 CD 8B AB 1E 49 1 9F 67 87 51 40 3E E0 CA
78 5F 75 BE 64 D8 5B 7B 14 4A 32 65 D 91 87
```

```
35 2F 1A 72 6B A7 A4 EB CA 7F DE 89 51 B4 3D BD D BD 24 9C
EB A4 2D 21 C8 C 11 CF 63 5F 82 93 D 80 14 A4 7F D6 85 40
1 6E 53 FE 41 5C A5 F3 E0 53 3F B0 97 6B 87 89 5F 1F 8F 30
CF D D 7C D3 57 25 E0 7C 2D E8 35 A9 6C D9 DA A1 5F D4 8B
3D D3 5E 43 64 C0 50 57 F3 72 F6 BB 87 A4 D6 D6 DD 8E 84 B0
DC A8 73 AA 58 69 25 8E 2A D3 A9 70 C 99 7 88 9E EE 44 E0
8C 5C 1 5D ED EF 81 5C FF F 26 EC A4 45 AE FA 35 BA 2C C4
6 72 D2 F4 83 E3 19 5D 78 90 CB B0 44 D0 F5 FA 68 55 C5 44
5E 8E DC D2 70 41 5C F3 8D A4 73 85 87 CA 88 29 9C D6 28 5C
4 FA 4C 1B C1 4 94 7 77 CC 28 40 C3 94 B F4 4D B9 67 1
F0 77 E8 68 44 85 5A 81 5F 51 F2 67 9F 97 F0 B8 58 7A 65 94
F6 1 24 50 5 48 CD 8B AB 1E 49 1 9F 67 87 51 40 3E E0 CA
78 5F 75 BE 64 D8 5B 7B 14 4A 32 65 D 91 87
```

Program 4.3

Both SAQ 5 and Example 4 are limited to moves of up to &100 bytes. On many occasions we need to move more than &100 bytes. Let's look at a move routine for larger blocks of non-overlapping data.

Example 5

A block move of this type would be used as a subroutine in a program. We can assume that, somewhere in the program, the number of bytes to be moved has been noted. Since the block is more than &FF bytes, the size of the block must be a two-byte number. Its high-byte is the number of whole pages to be moved and its low-byte is the number of items in the last, incomplete page. The routine we are going to use moves all the whole pages first and then does the last, incomplete page.

We have labelled the block size as 'pages' for its high-byte and 'bytes' for its low-byte.

The program is:

```
10 REM LARGE MOVE
20
30 MODE 3
40 HIMEM = HIMEM - 200
50 oldlow = &70
60 oldhigh = &71
70 newlow = &72
80 newhigh = &73
90 pages = &74
100 bytes = &75
110
120 FOR pass = 0 TO 3 STEP 3
130   PX = HIMEM
140   [
150     OPT pass
160
170     .large_move
180     LDX pages      \Put No. of pages to move into X
190     BEQ partpage  \If no whole pages, do part page
200
220     LDY #0        \Set byte counter
230
240     .nextbyte
250     LDA (oldlow),Y \Get byte from old block
260     STA (newlow),Y \Store in new block
270     INY            \Increment byte counter
280     BNE nextbyte  \If Y<>0 then move next byte of current
                        page
290     INC oldhigh    \Move to next page to move
300     INC newhigh    \Move to next page to receive data
310     DEX            \Decrement page counter
```

```

330 BNE nextbyte \Loop back to move next whole page
340
350 .partpage
360 LDY bytes \Get byte counter of last (incomplete)
           page
370 BEQ exit \If part page empty, then exit
380
390 .remnant
400 DEY \Decrement last page byte counter
410 LDA (oldlow),Y \Get byte from last page
420 STA (newlow),Y \Store it in new area
430 CPY #0 \Last byte done?
440 BNE remnant \If not, loop back
450
460 .exit
470 RTS
480 ]
490 NEXT pass
500
510 REM Test data to move &230 items from &3000 to &3500
520 ?&70 = 0
530 ?&71 = &30
540 ?&72 = 0
550 ?&73 = &35
560 ?&74 = &1
570 ?&75 = &10
580 items = ?&74 * 256 + ?&75
590
600 REM Place some random data in old block
610 PRINT ' "Test data put into old block:" '
620 FOR I = &3000 TO &3000 + items - 1
630 ?I = RND(26) + 64 : PRINT CHR$(?I) ;
640 NEXT
650
660 REM Move data
670 CALL large_move
680
690 REM Check the new block
700 PRINT ' "Data read back from new block:" '
710 FOR I = &3500 TO &3500 + items - 1
720 PRINT CHR$(?I);
730 NEXT
740 PRINT

```

Program 4.4

Test run

Test data put into old block:

```
JOWVPLDERPTHLEKKPUNGWRROWMDIETPXULYTGJYCYAGIRBBEHEJKYRJCTOFQENVGWMTZX  
APXJJENCQFQKROEEVWPRTVCZLRIIGHIOTOSUOTUDRNXLCAWMLPVLKFQCLRLKZURGNITIUHH  
BTGPEYCCQRNQFYZQKKTKTFWJXJEDQZHBZHXZHIMCSPWGKVVPDRHGPPVQCCJFCRHHHEPWK  
KZGJEHMMMOYCYCZOBKBFZTXJJFEVRFQLZPXAFFDMNDPYOEJVJIGUMJJEXIIC110QH
```

Data read back from new block:

```
JOWVPLDERPTHLEKKPUNGWRROWMDIETPXULYTGJYCYAGIRBBEHEJKYRJCTOFQENVGWMTZX  
APXJJENCQFQKROEEVWPRTVCZLRIIGHIOTOSUOTUDRNXLCAWMLPVLKFQCLRLKZURGNITIUHH  
BTGPEYCCQRNQFYZQKKTKTFWJXJEDQZHBZHXZHIMCSPWGKVVPDRHGPPVQCCJFCRHHHEPWK  
KZGJEHMMMOYCYCZOBKBFZTXJJFEVRFQLZPXAFFDMNDPYOEJVJIGUMJJEXIIC110QH
```

Program 4.4

SAQ 6

RAM test

Sometimes even the best micros break down. One part of the micro that you can test for yourself is the RAM. You just test each memory location by writing numbers into it and reading them back. Write a RAM test program that checks a given block of pages by filling the memory block with 0s and then reading back, then filling with 1's and reading back and so on until &FF.

Assignment D

1. Write a program to search a list of more than &100 items for a given character. The base address of the list is stored in &70 and &71. The first item in the list is the number of whole pages in the list and the second item in the list is the number of items on the last (incomplete) page of the list.
2. Write a program to find the largest element in a list of up to &100 characters. The base address of the list is stored at &70 and &71 and the first item in the list is the number of elements in the list.

Objectives of Unit 4

After studying this unit you should be able to recognise:

- ☐ The immediate mode.
- ☐ The Zero Page mode.
- ☐ The absolute mode.
- ☐ The implied mode.
- ☐ The absolute indexed mode.
- ☐ The relative mode.
- ☐ The accumulator mode.
- ☐ The indexed indirect mode.
- ☐ The indirect indexed mode.

Answers to SAQs

SAQ 1

The critical line is LDA &3000, X. Since X is the value currently in the X register, the range of values of X is 0 to &FF. So this method is limited to &100 items.

SAQ 2

```
10 REM TRANSFER
20
30 MODE 7
40 DIM Z%50
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80     OPT pass
90
100    .transfer
110    LDX #&32          \Initialise counter with No. of items to
                        move
120
130    .next
140    LDA &3000,X       \Load an item
150    STA &3800,X       \Store it in new location
160    DEX
170    BNE next
180
190    RTS
200  ]
210  NEXT pass
220
230 REM Test routine
240 PRINT ' "Test data put into old block:" '
250 FOR I = &3001 TO &3032
260   ?I = RND(&FF)
270   PRINT ; ?I ; " ";
280   NEXT
290
300 REM Move data
310 CALL transfer
320
330 REM Check new block
340 PRINT ' "Data read back from new block:" '
350 FOR I = &3801 TO &3832
360   PRINT ; ?I ; " ";
370   NEXT
380 PRINT
```

Program 4.5

Test run

>RUN

Test data put into old block:

```
173 167 84 76 94 221 219 135 169 102 184 208 200 228 151 104 87 154 173 50 56
242 132 96 51 80 229 89 161 59 108 55 192 72 48 53 27 196 112 235 177 92 226
231 226 136 2 207 199 108
```

Data read back from new block:

```
173 167 84 76 94 221 219 135 169 102 184 208 200 228 151 104 87 154 173 50 56
242 132 96 51 80 229 89 161 59 108 55 192 72 48 53 27 196 112 235 177 92 226
231 226 136 2 207 199 108
```

Program 4.5

Note that we have started at the top of the fifty (decimal) numbers (&3032) and worked down. This simplifies the program since we can use BNE instead of CMP to decide whether or not to loop back.

SAQ 3

X can be 0 to &FF. But if X exceeded &B9 in the example the effective address of LDA &70, X would go out of Zero Page. But LDA &70, X is a Zero Page instruction. This contradiction is resolved by the 6502 ignoring any part of an ostensible Zero Page address which exceeds &FF.

SAQ 4

To jump more than 127 bytes as a result of a branch instruction, you combine the branch with JMP which has unlimited range. e.g.

...

...

BEQ local

...

.local This label is only used as a jumping-off point for JMP.

JMP distant

i.e. BEQ branches to a nearby launch pad called here local which turns out to be nothing more than a JMP to distant. So, in effect, BEQ goes to distant however far away it is.

Another method is to leapfrog from branch instruction to branch instruction but this can only be done if all the instructions use the same flag:



Of course, you must make sure that other parts of your program don't accidentally fall into one of the later branch instructions.

SAQ 5

```

10 REM MOVE
20
30 MODE 7
40 HIMEM = HIMEM - 200
50 oldlow = &70
60 oldhigh = &71
70 newlow = &72
80 newhigh = &73
90 FOR pass = 0 TO 3 STEP 3
100 PX = HIMEM
110 [
120 OPT pass
130
140 .move
150 LDY #&60 \Initialise byte counter
160 DEY

180 .nextbyte
190 LDA (oldlow),Y \Get byte from old block
200 STA (newlow),Y \Store it in new block
210 DEY \Decrement byte counter
220 BPL nextbyte \Loop back if more bytes to move
230
240 RTS
250 ]
260
270 NEXT pass
280 REM Test routine
290 ?&70 = 0
300 ?&71 = &30
310 ?&72 = 0
320 ?&73 = &38
330 PRINT "Test data put into old block:"
340 FOR I = &3000 TO &305F
350 ?I = RND(255) : PRINT ; ?I ; " " ;
360 NEXT
370
380 REM Move data
390 CALL move
400
410 PRINT "Data read back from new block:"
420 FOR I = &3800 TO &385F
430 PRINT ; ?I ; " " ;
440 NEXT

```

Program 4.6

Test run

>RUN

Test data put into old block:

```
66 121 238 238 224 123 127 9 51 30 12 165 137 117 136 166 249 144 23 130 12
4 20 180 14 228 211 178 138 190 142 90 59 27 54 240 73 24 111 110 252 115 3
4 170 7 66 91 198 3 25 119 214 46 152 79 227 252 27 140 190 79 249 159 57 9
3 123 66 243 242 81 97 187 65 101 62 194 87 29 202 229 136 251 135 84 137 8
7 217 189 18 173 123 50 254 43 196 94 115
```

Data read back from new block:

```
66 121 238 238 224 123 127 9 51 30 12 165 137 117 136 166 249 144 23 130 12
4 20 180 14 228 211 178 138 190 142 90 59 27 54 240 73 24 111 110 252 115 3
4 170 7 66 91 198 3 25 119 214 46 152 79 227 252 27 140 190 79 249 159 57 9
3 123 66 243 242 81 97 187 65 101 62 194 87 29 202 229 136 251 135 84 137 8
7 217 189 18 173 123 50 254 43 196 94 115
```

Program 4.6

SAQ 6

```
10 REM RAM TEST
20
30 MODE 7
40 HIMEM      = HIMEM - 200
50 startlow   = &70
60 starthigh  = &71
70 endlow     = &72
80 endhigh    = &73
90 pages      = &74
100 character = &75
110 base      = &77
120
130 FOR pass = 0 TO 3 STEP 3
140   PX = HIMEM
150   [
160   OPT pass
170
180   .ramtest
190   LDA endhigh      \Get top page number
200   SEC
210   SBC starthigh    \Subtract bottom page number
220   STA pages        \Save number of pages to check
230   LDA starthigh    \Get high-byte of bottom page
240   STA base         \Save it in base
250
260   .fills
270   LDA #0
280   STA character     \Initialise check character
290
```

```

300 .nextvalue
310 JSR fill      \Fill memory block with current check
                  character
320 JSR check     \Read back memory block against current
                  check character
330 INC character \Move on to next check character
340 BNE nextvalue \If not zero, loop back for 'fill'/
                  'check' cycle

350 RTS
360
370 \SUBROUTINE TO FILL ALL THE PAGES
380 .fill
390 LDA base      \Get high-byte of base of memory block
400 STA starthigh \Use it to re-set high-byte of base of
                  memory block
410 LDA character \Get check character
420 LDX pages     \Initialise counter for 'pages checked'
430
440 .nextpage
450 LDY #0        \Initialise byte counter for current
                  page
460 .nextbyte
470 STA (startlow),Y \Put 'character' in a memory location
480 INY           \Move to next location
490 BNE nextbyte  \Loop back if page not finished
500
510 INC starthigh \Move to next page
520 DEX          \Decrement 'pages done' counter
530 BNE nextpage  \Loop back if more pages to do
540 RTS
550
560 \SUBROUTINE TO READ BACK ALL THE PAGES
570 .check
580 LDA base      \Get high-byte of base of memory block
590 STA starthigh \Use it to re-set high-byte of base of
                  memory block
600 LDX pages     \Initialise counter for 'pages checked'
610
620 .nextpageread
630 LDY #0        \Initialise byte counter for current page
640
650 .nextbyteread
660 LDA (startlow),Y \Read 'character' in a memory location
670 CMP character  \Is it the same as the current check
                  character?
680 BNE failed     \If not, then exit. X and Y registers
                  hold position of failed byte

```

```

690 INY                \Ils move to next byte
700 BNE nextbyteread  \and loop back
710
720 INC starthigh      \Move to next page
730 DEX                \Decrement page read
740 BNE nextpage       \If not finished, loop back to do next
                        page
750 RTS
760
770 .failed
780 RTS
790 ]
800 NEXT pass
810
820 INPUT "Enter start page No. in decimal" S%
830 ?&71 = S%
840 INPUT "Enter end page No. in decimal " E%
850 ?&73 = E%
860
870 result = USR(ramtest)
880 IF (result AND &FFFF) DIV &100 = 0 THEN PRINT "Memory
    OK." ELSE PRINT "Failed at page &" ; (result AND &FFFF) DIV
    &100 " byte &" ; (result AND &FFFFFF) DIV &10000

```

Program 4.7

UNIT 5

Multiplication and division

- 5.1 The problem of multiplication
- 5.2 The Shift and Rotate instructions
- 5.3 Assembly language multiplication
- 5.4 8-bit binary division
- 5.5 INC and DEC
- 5.6 Assembly language 8-bit division

Assignment E

Objectives of Unit 5

Answers to SAQs

5.1 The problem of multiplication

6502 assembly language has no command for multiply or divide. All it provides for arithmetic are the ADC and SBC instructions which we have already met. (Although, as you will see, ASL effectively multiplies by 2 and LSR effectively divides by 2). This means that all further arithmetic must be expressed in terms of these four instructions. (The manufacturer of a microcomputer will often provide his own subroutines for arithmetic which can be called through specified locations. These are not discussed here since this section is confined to pure assembly language methods which will work with any 6502 system). So we start this unit by looking at a means of breaking down multiplication into a series of additions. The method is one which you will have met before – long multiplication.

The 6502 is going to work in binary but we will first look at base 10 long multiplication. Consider the sum $873 * 123$.

$$\begin{array}{r}
 873 \\
 123 \\
 \hline
 \text{Partial products) } 2619 \quad (= 873 * 3) \\
 1746 \quad (= 873 * 20) \\
 873 \quad (= 873 * 100) \\
 \hline
 107379
 \end{array}$$

That involved multiplication by 1, 2 and 3 (the digits of 123). Other decimal long multiplication could involve multiplications by 1, 2, 3, 4, 5, 6, 7, 8, or 9. But in binary long multiplication we only need to multiply by 1. That is, all multiplications become additions. Consider $13 * 7$ when done in binary.

$$\begin{array}{r}
 1101 \\
 111 \\
 \hline
 \text{Partial products) } 1101 \quad (= 1101 * 1) \\
 1101 \quad (= 1101 * 10) \\
 1101 \quad (= 1101 * 100) \\
 \hline
 1011011
 \end{array}$$

Next consider $13 * 5$.

$$\begin{array}{r}
 1101 \\
 101 \\
 \hline
 1101 \quad (= 1101 * 1) \\
 0000 \quad (= 1101 * 0) \\
 1101 \quad (= 1101 * 100) \\
 \hline
 1000001
 \end{array}$$

From this you can see that long multiplication involves successive left shifts of the multiplicand followed by addition of the shifted multiplicand to the partial products when the multiplier has a 1 in the relevant position. In the standard method of long multiplication illustrated above, we listed all the partial products and then added them together at the end. When using a computer it is easier to accumulate the partial products as we go along. Here is our first decimal example, $873 * 123$ done by the method of accumulating the partial products:

	Accumulated partial product
Start	0
First partial product = $873 * 3 = 2519$	$0 + 2519 = 2519$
Second partial product = $873 * 20 = 17460$	$2519 + 17460 = 20079$
Third partial product = $873 * 100 = 87300$	$20079 + 87300 = 107379$

A flowchart illustrating this method for binary multiplication is shown in Figure 5.1. We have assumed that the multiplier is 8 bits long.

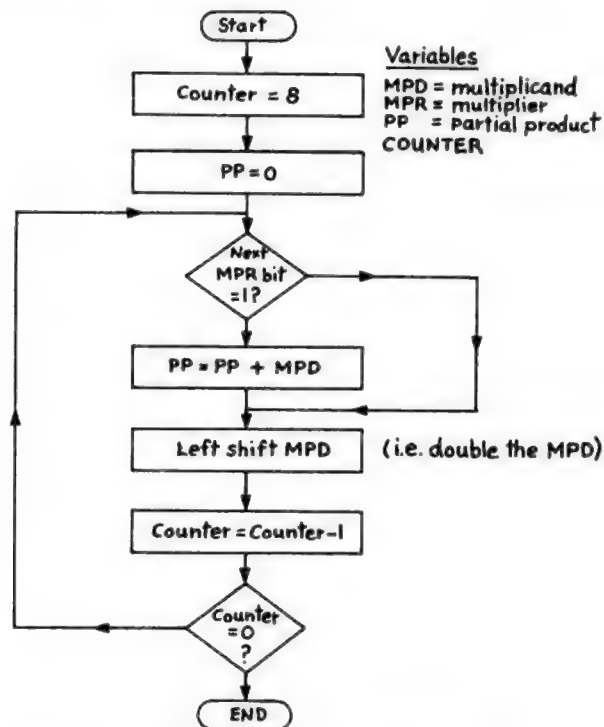


Figure 5.1 Multiplication of two 8-bit numbers

SAQ 1

Multiply the binary numbers 1010 by 1101 (a) by standard long multiplication (b) by the method in Figure 5.1. In the case of (b), note down all the stages of the accumulation of the partial products.

If you examine Figure 5.1 closely, you will see that it still cannot be programmed in the 6502 language that you have met so far. To program Figure 5.1 we need to be able to test successive bits of the multiplier to see if they are 0 or 1. We also need to be able to shift the multiplicand left by one bit position. Both these activities can be accomplished with some new 6502 instructions which we shall now introduce.

5.2 The Shift and Rotate instructions

There are four instructions which shift the bits of a register or memory location. In each case the bit that is pushed out falls into the carry flag location. First, the two shift instructions.

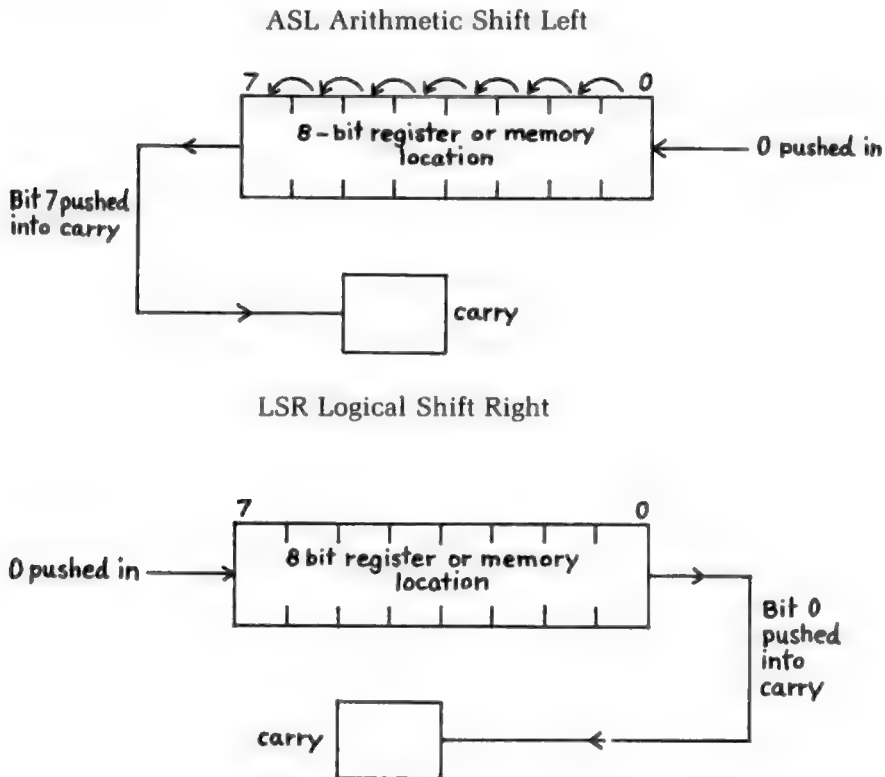


Figure 5.2 The two shift instructions

Both ASL and LSR result in the loss of one bit although it is available in the carry bit for inspection or 'catching' for some other purpose. Thus if the accumulator contained 10110110 the effect of LSR on the accumulator would be:

	Accumulator	
Before	10110110	
LSR A		
After	01011011	0 pushed off into carry.
	0 appears here	

The effect of a left shift on a number is to multiply that number by 2. The effect of a right shift is to divide the number by 2, losing the remainder.

The situation is slightly different with the two rotate instructions. These are like the shift instructions except that the carry bit is pushed back into the other end of the memory location or register being rotated. This is illustrated in the following two figures.

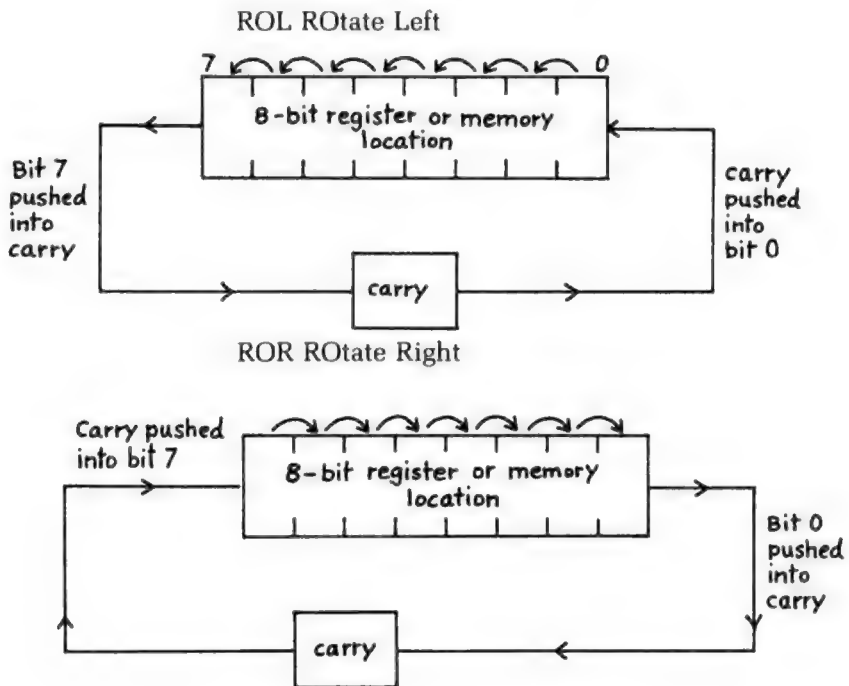


Figure 5.3 The two rotate instructions

The effect of ROR on 10110110 is shown below. In order to illustrate ROR, we must say what value the carry has before executing the command ROR since the carry bit is going to become part of the accumulator.

	Carry	Accumulator
Before	1(say)	10110110
ROR		
After	0	11011011

A useful feature of LSR (and ROR with $C = 0$) is that it performs division by 2 with any remainder being discarded. (This is the same as BASIC's DIV 2).

The BBC assembler, in common with many other assemblers, insists that you specify the operand with the shift and rotate instructions. You must say ASL A, LSR A, ROL A, ROR A (not just ASL, LSR, ROL, ROR) when shifting or rotating the accumulator.

SAQ 2

Write a program to find how many 1's a number has. Put the number in location &70. Place the answer in &80.

5.3 Assembly language multiplication

In Figure 5.1 we showed a method of performing binary multiplication. In 5.2 we introduced the shift instructions needed for this multiplication. However, even if we limit the multiplier and multiplicand to 8 bits, programming the method in Figure 5.1 presents two problems. First, the cumulated partial product may exceed 8 bits. Second, the left shifted multiplicand will lose bits as they are shifted out of the register holding the multiplicand. Let's look again at Figure 5.1 to see if we can avoid shifting the multiplicand at all.

The method used in Figure 5.1 was to:

- ☐ left shift the multiplicand
- ☐ add to the cumulated partial product

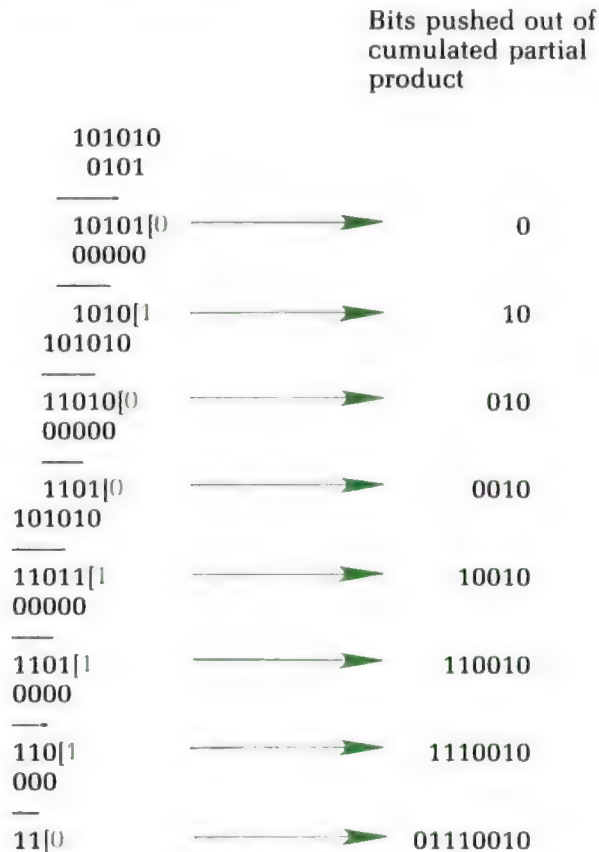
If you look back again at the answer to SAQ 1(b), you will see that the above method is exactly the same in effect as:

- ☐ add the multiplicand to the cumulated partial product
- ☐ right shift the cumulated partial product

SAQ 3

Re-work SAQ 1 by this new method. Collect the right shifted bits that fall off the partial product in a new location. (Do this as a paper and pencil multiplication and not as an assembly language program).

The result of SAQ 3 shows that we can perform binary multiplication without having to shift the multiplicand. Instead, we shift the cumulated partial product bit by bit into a new memory location which at the end of the multiplication contains the full answer. But the answer could be more than 8 bits long. If that were the case, the 8 shifted bits of the partial product would be the 8 *least* significant bits. The 8 most significant bits would still be in the partial product location. Although SAQ 3 appears to work, in practice we must allow for a 16-bit result. If this is difficult to follow, consider this case:



The total is 11 01110010 = 882. The least significant part is 01110010 which was shifted out into the result location but the most significant digits, 11, remained in the partial products sum.

SAQ 4

Multiply 1111 by 1011 using the above method. Establish which bits form the least significant byte and which bits form the most significant byte.

We can now set out the flowchart for assembly language multiplication of two 8-bit numbers with a possible 16-bit result. This is shown in Figure 5.4.

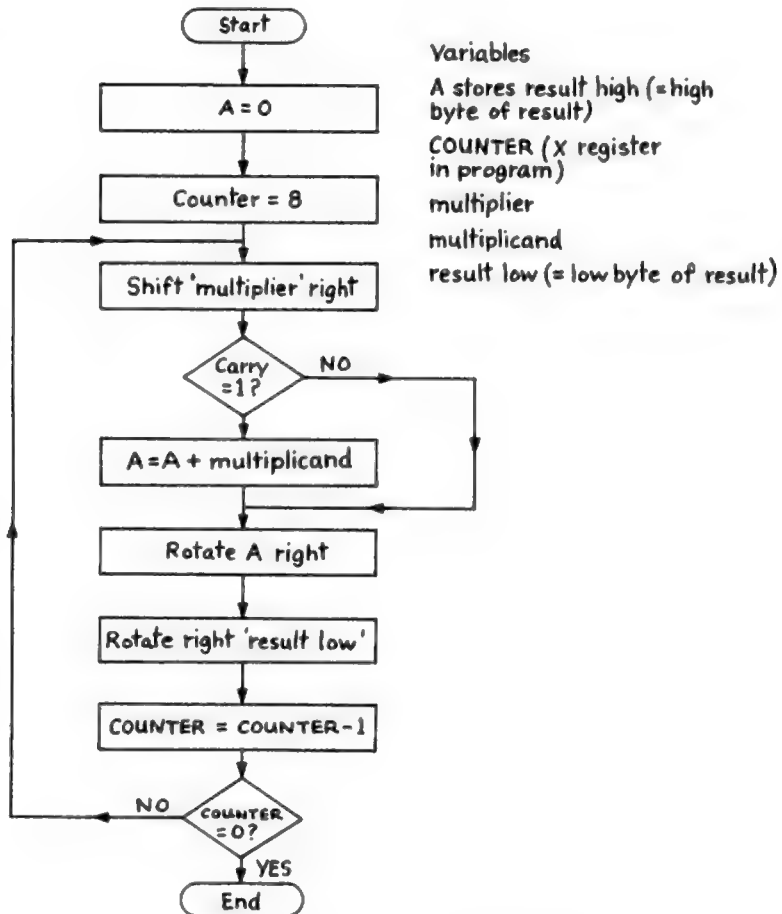


Figure 5.4 Flowchart for 8 bit multiplication

And, finally, the program which is Program 5.1.

The program assumes that the multiplicand is in &70 and the multiplier in &71. The low-byte of the answer is stored in &72 and the high-byte is transferred into location &73 at the end of the program.

```

10 REM 8 BIT MULTIPLICATION
20
30 DIM Z%50
40 multiplicand = &70
50 multiplier   = &71
60 resultlow    = &72
70 resulthigh   = &73
80
90 FOR pass = 0 TO 3 STEP 3
100  PX = Z%
110  [
120  OPT pass
130
140  .multiply
150  LDA #0           \Set high-byte of answer to zero
160  LDX #8           \Set bit counter
170
180  .next
190  LSR multiplier   \Shift next multiplier bit into carry
200  BCC rot
210  CLC
220  ADC multiplicand \Add multiplicand
230
240  .rot
250  ROR A           \Shift lsb of high-byte into carry
260  ROR resultlow    \Catch carry in low-byte of answer
270  DEX             \Decrease counter
280  BNE next         \Repeat if more bits
290  STA resulthigh   \Store high-byte
300
310  RTS
320  ]
330  NEXT pass
340
350 REM Test run
360 INPUT "First number " first
370 INPUT "Second number " second
380 ? &70 = first
390 ? &71 = second
400
410 CALL multiply
420
430 PRINT "Product low-byte = " ? &72
440 PRINT "Product high-byte = " ? &73

```

Program 5.1

Test runs

First a test of multiplication that should give an 8-bit result: $12 * 5$.

```
>RUN
First number 12
Second number 5
Product low-byte =      60
Product high-byte =      0
```

Second a test that should give a 16-bit result: $137 * 231$.

```
>RUN
First number 137
Second number 231
Product low-byte =     159
Product high-byte =    123
```

The total is $123 * 100 + 159 = 31647$, which is the correct result.

☒ Load Program 5.1. Check that your results agree with those above. Then try other multiplications.

Program 5.1 covered the multiplication of two 8-bit unsigned numbers. More complicated routines are needed if the numbers are signed or larger than 8-bits. We shall not deal with multiplying signed numbers in this book although we shall deal with 16-bit unsigned multiplication later in this book. First, however, let's look at 8-bit division.

5.4 8-bit binary division

Binary multiplication involved successive additions. Binary division involves successive subtractions of the divisor from the dividend. Consider $10111101 / 101$ ($189 / 5$):

```

  / 00100101
101 10111101
  -101
  ----
    0111
   -101
   ----
    1001
   -101
   ----
    100
```

The answer is 100101 (base 2), remainder 100 (base 2) i.e. (decimal 35, remainder 4).

As you might expect, the most efficient division program does not come from simply programming the above routine. Instead, we shift successive bits of the dividend into the accumulator until we can subtract the divisor from the accumulator. Then we add 1 to the quotient because the quotient counts how many times we have been

able to subtract the divisor from the dividend. This continues until all 8 bits of the dividend have been shifted. Every shift of the dividend must be accompanied by a compensating shift of the quotient since we are subtracting multiples of the divisor. Let's look at this method by dividing 10111 by 101.

Stage	Accumulator	Dividend	Quotient
Start	00000000	00010111	00000000
Shift 1	00000000 Can't subtract 101	← 00101110	00000000
Shift 2	00000000 Can't subtract 101	← 01011100	00000000
Shift 3	00000000 Can't subtract 101	← 10111000	00000000
Shift 4	00000001 Can't subtract 101	← 01110000	00000000
Shift 5	00000010 Can't subtract 101	← 11100000	00000000
Shift 6	00000101 Can subtract 101	← 11000000	00000000
Subtract	00000000	← 11000000	00000001 (Add 1 to quotient)
Shift 7	00000001 Can't subtract 101	← 10000000	00000010
Shift 8	00000010 Can't subtract 101	← 00000000	00000100

After 8 shifts, the dividend register is empty. The quotient register contains 100 and there is 10 left in the accumulator. So:

$$10111 / 101 = 100, \text{ remainder } 10.$$

SAQ 5

Divide the binary number 110110 by the binary number 111 using the above method. Show the contents of the accumulator, dividend register and quotient at each step.

If you inspect the register containing the dividend in our example or in your answer to SAQ 5, you will see that each successive shift creates one more free location (shaded in our example) in the dividend register. These locations are 'free' and don't affect the computation. At the same time these free bit places are always enough to hold the quotient at that stage of the computation. We can therefore get rid of the quotient as a separate store and simply save the quotient by adding 1 to the dividend register whenever the quotient needs a 1 adding to it. (The

0's will appear automatically as part of the shift routine). Here, then, is our division example re-worked in this way.

Stage	Accumulator	Dividend/quotient	
Start		00010111	
Shift 1	00000000 ←	0010111 0 D Q	
Shift 2	00000000 ←	010111 D Q	
Shift 3	00000000 ←	10111 000 D Q	
Shift 4	00000001 ←	0111 0000 D Q	
Shift 5	00000010 ←	111 00000 D Q	
Shift 6	00000101 ←	11 000000 D Q	
Subtract /	00000000 ←	11 000001 D Q	(Note 1 added to quotient)
Shift 7	00000001 ←	1 0000010 D Q	(Note quotient shift)
Shift 8	00000010 ←	00000100	
	Remainder	Quotient	

SAQ 6

Repeat SAQ 5 using a dividend/quotient register instead of two registers. Show at each stage where the 'boundary' between dividend and quotient is in the single register.

We now have an algorithm for 8-bit binary division. To program it we need one new instruction.

5.5 INC and DEC

In Unit 3 you met INX, INY, DEX and DEY which we used to increment or decrement the X and Y registers by 1. There are equivalent instructions to increment and decrement a memory location.

INC INCrement memory location by 1
 DEC DECReament memory location by 1

INC &75 will make the contents of &75 1 larger. DEC &75 will reduce the contents by 1. We shall need INC for the binary division program, since we need to add 1 to the quotient every time we subtract the divisor. The great value of INC and DEC is that they allow us to alter a memory location without disturbing the contents of the accumulator. (You can't, however, INC or DEC the accumulator.)

5.6 Assembly language 8-bit division

The program assumes that the divisor and dividend are in memory locations &70 and &71. &71 is going to be the dividend/quotient register. At the end of the program run, &71 will contain only the quotient. The bits of the dividend are rotated into the accumulator which, at the end of the run, will contain the remainder. As we don't want to leave a result in the accumulator, we transfer it to &73.

```

10 REM 8 BIT DIVISION
20
30 DIM Z%50
40 dividend = &71
50 divisor = &70
60 remainder = &73
70
80 FOR pass = 0 TO 3 STEP 3
90   P% = Z%
100  [
110   OPT pass
120
130   .division
140   LDA #0           \Clear A
150   LDX #8           \Set shift counter
160   SEC              \Set carry before a low byte subtraction
170
180   .shift
190   ASL dividend     \Shift dividend left
200   ROL A            \Rotate to catch carry bit in A
210   CMP divisor      \Is dividend large enough to subtract
                        divisor?
220   BCC next         \If not, skip next two instructions
230   SBC divisor       \Subtract divisor from A
240   INC dividend     \Add 1 to quotient
250
260   .next
270   DEX              \Decrement shift counter
280   BNE shift        \Loop back if more shifts
290   STA remainder    \Store remainder

```



```

300
310   RTS
320   J
330   NEXT pass
340
350 REM Test run
360 INPUT "Enter first number " first
370 INPUT "Enter second number " second
380 ? &71 = first
390 ? &70 = second
400
410 CALL division
420
430 PRINT "Result = " TAB(15) ; ? &71
440 PRINT "Remainder = " TAB(15) ; ? &73

```

Program 5.2

First test run

```

>RUN
Enter first number 12
Enter second number 145
Result =          0
Remainder =       12

```

Second test run

```

>RUN
Enter first number 145
Enter second number 12
Result =          12
Remainder =        1

```

☒ Try Program 5.2 with different values.

Assignment E

1. Quite often we want to multiply a number by 10 (decimal). Write a program to do this. Make use of the fact that $10 * n$ is the same as $8 * n + 2 * n$.

Objectives of Unit 5

After studying this Unit you should be able to:

- ☐ Perform binary multiplication with pencil and paper.
- ☐ Use ASL and LSR.
- ☐ Use ROL and ROR.
- ☐ Use INC and DEC.
- ☐ Write an 8-bit multiplication program.
- ☐ Perform binary division with pencil and paper.
- ☐ Write an 8-bit division program.

Answers to SAQs

SAQ 1

(a)

```

    1010
    1101
    ----
    1010
    0000 ← This line could be omitted.
    1010
    1010
    -----
    10000010
```

Check: $10 * 13 = 130$.

(b)

```

PP = 0
PP = 1010
PP = 1010
PP = 1010 + 101000 = 110010
PP = 110010 + 1010000 = 10000010
PP = 10000010
PP = 10000010
PP = 10000010
PP = 10000010
PP = 10000010
```

SAQ 2

```

10 REM BIT COUNT
20
30 DIM Z%50
40
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80   OPT pass
90
100  .bitcount
110  LDY #0      \Set counter for the 1's
120  LDX #7      \Set counter for the bits
130
140  .rotate
150  ROR &70     \Put a bit into carry
160  BCC next    \If the bit was a 1, skip the next
                  instruction
170  INY         \Otherwise, increment the bit counter
180
190  .next
200  DEX
```

```

210 BPL rotate \Loop back if more bits to check
220
230 .store
240 STY &80
250 RTS
260 J
270 NEXT pass
280
290 REM Test routine
300 INPUT "Enter number for bit count " number
310 ?&70 = number
320
330 CALL bitcount
340
350 PRINT\; number " contains " ; ?&80 " bits set to 1."

```

Test runs

```

>RUN
Enter number for bit count 255
255 contains 8 bits set to 1.

```

```

>RUN
Enter number for bit count 0
0 contains 0 bits set to 1.

```

```

>RUN
Enter number for bit count 123
123 contains 6 bits set to 1.

```

Program 5.3

SAQ 3

Right shifted bits
collected in a new location

	PP = 0	
	PP = 0 + 1010	
Right shift PP	PP = 101[0] →	0
	PP = 0 + 101	
Right shift PP	PP = 10[1] →	10
	PP = 10 + 1010 = 1100	
Right shift PP	PP = 110[0] →	010
	PP = 1010 + 110 = 10000	
Right shift PP	PP = 1000[0] →	0010
	PP = 0 + 1000 = 1000	
Right shift PP	PP = 100[0] →	00010
	PP = 0 + 100 = 100	
Right shift PP	PP = 10[0] →	000010
	PP = 0 + 10 = 10	
Right shift PP	PP = 1[0] →	0000010
	PP = 0 + 1 = 1	
Right shift PP	PP = 0[1] →	10000010

After 8 right shifts, the answer, 10000010, has appeared in the new location and we haven't had to shift the multiplicand at all.

SAQ 4

11111		
10111		

11111[1] →		1
11111		

10111[0] →		01
11111		

11011[0] →		001
00000		

1101[1] →		1001
11111		

10110[0] →		01001
00000		

1011[0] →		001001
0000		

101[1] →		1001001
000		

10[1] →		11001001

Most significant byte = 10. Least significant byte = 11001001.

Check:

11111 = 31
 10111 = 23
 31 * 23 = 713

$$10\ 11001001 = 512 + 128 + 64 + 8 + 1 = 713$$

SAQ 5

Stage	Accumulator	Dividend	Quotient
Start	00000000	00110110	00000000
Shift 1	00000000 Can't subtract 111	01101100 ←	00000000 ←
Shift 2	00000000 Can't subtract 111	11011000 ←	00000000 ←
Shift 3	00000001 Can't subtract 111	10110000 ←	00000000 ←

Shift 4	00000011 ←	01100000 ←	00000000
	Can't subtract 111		
Shift 5	00000110 ←	11000000 ←	00000000
	Can't subtract 111		
Shift 6	00001101 ←	10000000 ←	00000000
	Subtract 111		Add 1 to quotient
	00000110 ←	10000000 ←	00000001
Shift 7	00001101 ←	00000000 ←	00000010
	Subtract 111		Add 1 to quotient
	00000110	00000000	00000011
Shift 8	00001100 ←	00000000 ←	00000110
	Subtract 111		Add 1 to quotient
	00000101	00000000	00000111

Hence $110110 / 111 = 111$, remainder 101.

Check: $54 / 7 = 7$, remainder 5.

SAQ 6

Stage	Accumulator	Dividend/quotient
Start	00000000	00110110
		D
Shift 1	00000000 ←	0110110 0
		D q
Shift 2	00000000 ←	110110 00
		D q
Shift 3	00000001 ←	10110 000
		D q
Shift 4	00000011 ←	0110 0000
		D q
Shift 5	00000110 ←	110 00000
		D q
Shift 6	00001101 ←	10 000000
		D q
	Subtract 111	Add 1 to quotient
	00000110	10000001
		D q
Shift 7	00001101 ←	0 0000010
		D q
	Subtract 111	Add 1 to quotient
	00000110	0 0000011
		D q
Shift 8	00001100 ←	00000110
		q
	Subtract 111	Add 1 to quotient
	00000101	00000111
	Remainder	Quotient

UNIT 6

Lists and tables

- 6.1 Assembly language lists and tables
- 6.2 Inputting a sentence
- 6.3 Printing a list
- 6.4 Sorting a list
- 6.5 Searching an ordered list
- 6.6 Inserting an item into a list
- 6.7 Putting lists into memory
- 6.8 Look-up tables

Assignment F

Objectives of Unit 6

Answers to SAQs

6.1 Assembly language lists and tables

In BASIC programs, data frequently comes in the form of lists and tables. Even when it doesn't quite come in such a form, we may find a way of putting it into such a form. We do that because BASIC provides 'data structures' within its language e.g. lists (one dimensional arrays) and tables (two dimensional arrays). In assembly language, no such data structures are provided, so we have to create our own. We shall look at some very simple data structures and some routines for handling them. In particular we shall look at how to create lists and then search, sort, add to and delete from them.

Before we start, let's first illustrate the complexity of having to create your own data structures by comparing a simple task for BASIC with its equivalent in assembly language.

Example 1

Write a BASIC program to input 10 names and store them in a suitable data structure.

Solution

```
10 DIM N$(10)
20 FOR I = 1 TO 10
30   INPUT "NEXT NAME " N$(I)
40   NEXT I
```

```
RUN
NEXT NAME PETER
NEXT NAME JONATHAN
NEXT NAME MARY
```

etc.

That was easy. So easy that you have probably not bothered to consider what the computer has done. First it has provided a structure to locate 10 strings. That structure was the array N\$(10). It did this even though it knew nothing about each individual string that was to be stored. Second, as we input each name, it was stored in its appropriate place, large enough to accommodate it even though the names are of varying lengths. Third, if we want to retrieve a name, we can do so just by knowing its position in the list e.g. N\$(3) = "MARY".

Now consider what would be involved in doing that in assembly language.

First, assembly language has no arrays and no string handling commands. So to store the name PETER, we have to store it as ASCII code:

50, 45, 54, 45, 52 i.e. PETER

or

50, 65, 74, 65, 72 i.e. Peter

or

70, 65, 74, 65, 72 i.e. peter

Second, we must decide *where* to store the name. We have to tell the computer which memory locations to use.

Third, we have to know where in the memory the name PETER ends and the name JONATHAN begins. We could do this by putting an 'end of item' marker after each name (e.g. a 'carriage return') so the list would appear in memory as:

```
50 45 54 45 52 0D 4A 4F 4E 41 54 48 41 4E 0D
P E T E R   J O N A T H A N
```

Or, we could allocate, say, 10 memory locations to each name and enter names as:

```
PETER.....JONATHAN...MARY.....
```

In that way, we would always know where to find the next name or the *n*th name but we have to make sure that no name is longer than 10 characters.

So there is clearly a lot more to consider about storing data in assembly language programming than in a high level language. We shall work with very simple data as an introduction to data handling. We start with inputting a sentence and storing it as a list of characters in memory.

6.2 Inputting a sentence

We are going to input the data at the keyboard directly into the memory locations we have specified in our program. The BBC Micro has a built-in routine for entering single characters which is called through &FFE0. The name of the routine is OSRDCH.

OSRDCH Operating System Read Character
This is called through &FFE0.
It accepts one character from the keyboard and
puts its ASCII value in the accumulator.

When you want to take a character from the keyboard you write

```
JSR &FFE0
```

or, better (because it makes your program more readable):

```
JSR osrdch
```

having previously defined &FFE0 = oswrch at the start of the program.

OSRDCH fetches a character from the keyboard and puts it into the accumulator. But it *only* does that, so the character is not displayed on the screen as it would be with the BASIC statement INPUT. (OSRDCH is therefore identical to GET in BASIC). So if you want to see what you are entering as you go along you need to use OSWRCH immediately to print each character as it is accepted from the keyboard. We will demonstrate this method in the next example.

(This is a simplified explanation of OSRDCH. It also contains a system for detecting errors during input and for detecting whether or not Escape was pressed. If Escape was pressed, this has to be acknowledged. We have omitted this detail in Program 6.1 but the method of acknowledging Escape is dealt with in section 8.7).

Example 2

Write a program to input up to &100 one-byte items from the keyboard and store them in memory starting at &3000.

Solution

There are many ways in which such lists can be stored. The simplest puts the items (7, say) in addresses &3000 to &3006 and then puts an 'end of list' item in &3007. This 'delimiter' would probably be 0 since 0 has no character associated with it in ASCII code. The second, and slightly more complicated method, is to store the list in addresses &3001 to &3007 and then to store the length of the list (7) in &3000 i.e. back at the start of the list. These two methods are illustrated below, both storing **MACBETH**

Memory locations								
	&3000	&3001	&3002	&3003	&3004	&3005	&3006	&3007
Method 1	&40	&41	&43	&42	&45	&54	&48	&0
Method 2	&07	&40	&41	&43	&42	&45	&54	&48

It is up to you to decide which method you use for any particular program. As a guide, the delimiter method is best for lists that will only be worked through sequentially. On the other hand, if a list is destined to be worked through in some other way (e.g. with certain types of search routine), then the length of the list needs to be known at the outset. Storing the length at the start of the list provides this information. Because this method leads to lists which can be more easily manipulated, it is the one mostly used in this Unit.

To work through the list, we need to know both its base address and its length. But we don't want the input program to work for only one base address. We will use indirect indexed addressing (from section 4.12) so that the base address can be stored at a fixed location. We shall also need to arrange for the program to count the number of items input, detect an end-of-list character and to store the count length at the beginning of the list. Then whenever we access the list, we shall know that the first item is the length of the list and not an item of data. A flowchart for this method is shown in Figure 6.1.

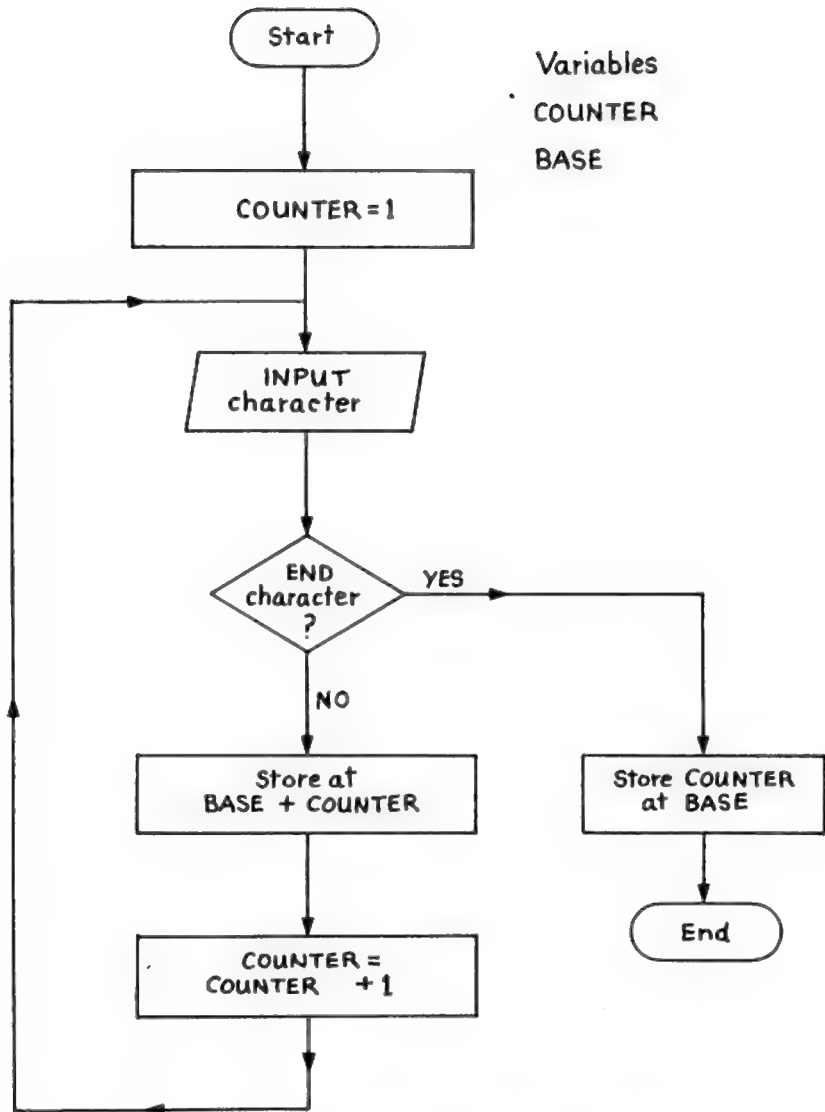


Figure 6.1 Flowchart for inputting a list such as a sentence

We shall use 'return' (ASCII &0D) as an end character. The address of the base is stored at &70 and &71 (the base will have a two-byte address). So the program starts with

```

listloco = &00
listlochi = &30

```

If, later, you wanted to use the program for a list with a base at another address, only these two numbers would have to be changed.

The program is given in Program 6.1.

```
10 REM INPUT
20
30 MODE 7
40 listloclo = 80
50 listlochi = 830
60 base = 870 : REM Plus 871
70 eol = 800
80 osrdch = &FFED
90 oswrch = &FFEE
100
110 DIM Z%100
120 FOR pass = 0 TO 3 STEP 3
130   P% = Z%
140   [
150     OPT pass
160
170     .input
180     LDA #listloclo \Get low-byte of address to store list
190     STA 870        \Store in zero Page
200     LDA #listlochi \Get high-byte of address to store list
210     STA 871        \Store in zero Page
220     LDY #1         \Set character counter
230
240     .next
250     JSR osrdch      \Get next character
260     JSR oswrch      \Print character on screen
270     CMP #eol       \Is it 'return'?
280     BEQ length      \If yes, branch & store length
290     STA (base),Y    \Store character in memory
300     INY
310     BNE next        \Loop back for next character if <
                       254 characters input
320
330     .length
340     DEY             \Adjust the counter to give length of list
350     TYA             \Put length in A
360     LDY #0          \Set pointer for base of list
370     STA (base),Y    \Store length at start of list
380
390     RTS
400   ]
410   NEXT pass
420
430 REM Test routine
```

```

440 CALL input
450
460 REM Print out the contents of the memory block
470 length = ?&3000
480 PRINT ' ' "Contents of &3001 to " ; "&3000 + length ":" '
490 FOR N = &3001 TO &3000 + ?&3000
500   PRINT CHR$(?N) ;
510   NEXT
520 PRINT

```

Program 6.1

- 240-310 This is the loop which accepts each character from the keyboard, tests whether it is the end character and, if not, stores it in memory at line 290.
- 330-370 Puts the length of the list in memory at the beginning of the list.
- 330 At line 330, Y is 1 greater than the length so it has to be decremented at line 340. This new value is the length of the list which is put into the accumulator at line 350. Then Y is set to 0 (line 360) so that the length can be stored at the beginning of the list (line 370).

Test run

```

>RUN
THIS SENTENCE WAS PUT INTO MEMORY AT &3000

Contents of &3001 to 302A:

THIS SENTENCE WAS PUT INTO MEMORY AT &3000
>

```

☒ Load Program 6.1 and test it with a sentence of your own.

SAQ 1

Write a version of Program 6.1 but using 0 as an end of list delimiter. (The list it creates will not need to have its length stored with it).

SAQ 2

Write a simplified version of Program 6.1 using absolute indexed X mode to transfer the characters to memory.

6.3 Printing a list

Now that we have developed a routine for getting a list into memory, we need a routine for printing out the list using assembly language instead of BASIC. Once we have that, we can develop routines for manipulating lists, knowing that at any stage we can print out the list to see what has happened to it.

Since the lists may be lengthy, we shall print them across the screen with a space between each character:

a b c d e ...

The routine for doing this for a list with its length stored in the first byte is shown in Figure 6.2.

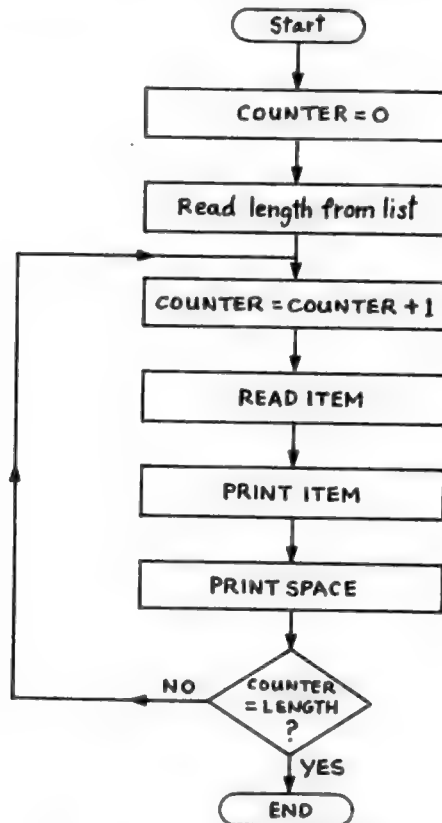


Figure 6.2 Printing out a list

SAQ 3

Write a program to implement Figure 6.2.

☒ Use your program to print the list you stored in memory in SAQ 2.

6.4 Sorting a list

The method we are going to use to sort a list into alphabetical or numerical order is the 'bubble' sort. This involves working through the list comparing adjacent items. If they are in the correct order, they are left alone. If they are in the wrong order, they are interchanged. After the first pass through the list, the largest item will be at one end. After the second pass, the next largest item goes to the second highest position, and so on. This is the commonest method of sorting and you may have met it already. In BASIC it is accomplished by nested FOR ... NEXT loops which sort an array such as X%(N) as shown in Program 6.2.

```
6.2. 10 REM BASIC BUBBLE SORT
      20
      30 FOR K% = 1 TO N - 1
      40   exchange = FALSE
      50   FOR L% = K% + 1 TO N
      60     IF X%(L%) < X%(K%) THEN T% = X%(L%) : X%(L%) = X%(K%)
      : X%(K%) = T% : exchange = TRUE
      70   NEXT L%
      80   IF exchange THEN NEXT K%
```

Program 6.2

For assembly language sorting, the list will be of the form shown in Figure 6.3. We go through the list comparing adjacent items and swapping them if they are in the incorrect order. This is done until a complete pass through the list produces no exchanges. When this is the case the list will be fully sorted. To detect this situation we use a 'flag' variable which is set to 0 before each pass through the list. If an exchange occurs, the flag is set to -1.

Because the bubble sort 'bubbles' the highest misplaced item up to its correct position, the unsorted part of the list becomes one item shorter after each full pass. So we can reduce the length of list that we search after each full pass and so speed up the sort. In assembly language, the bubble sort program follows a similar form, except that as there are no array structures, we use indirect indexed commands to read our unsorted items.

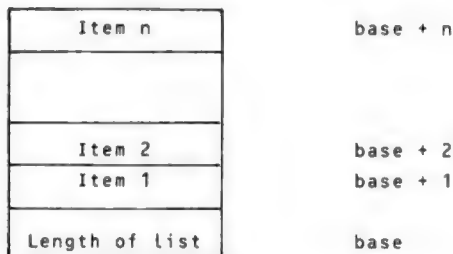


Figure 6.3 Unsorted list in memory

The flowchart is shown in Figure 6.4.

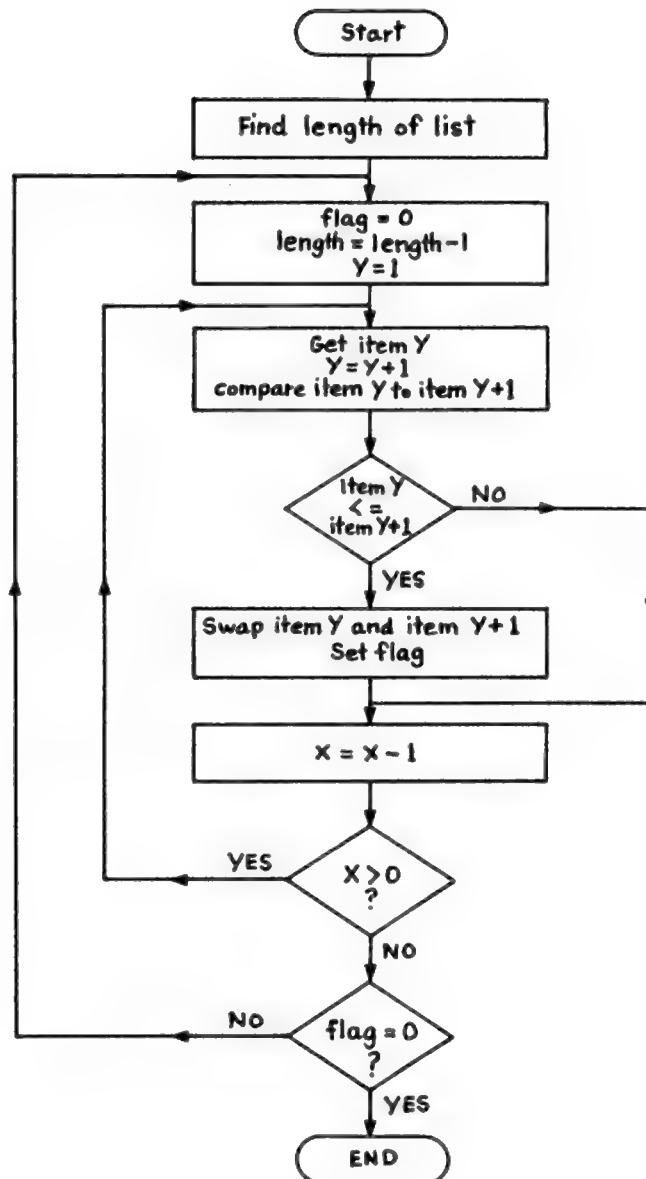


Figure 6.4 Flowchart for bubble sort

Program memory locations

base	The base address of the list	&70/1
len	Length of the list	&72
flag	Flag to indicate whether a swap has occurred	&73
temp	Temporary zero page store for the swap routine	&74

Notice that in lines 160 and 180 we have used a convenient method for extracting the low and high bytes of a 2-byte address. `xxxx MOD &100` extracts the low-byte and `xxxx DIV &100` extracts the high-byte.

```
10 REM BUBBLE SORT
20
30 MODE 3
40 location = &3000
50 base = &70
60 len = &72
70 flag = &73
80 temp = &74
90 DIM Z%100
100 FOR pass = 0 TO 3 STEP 3
110   PX = Z%
120   [
130     OPT pass
140
150     .sort
160     LDA #location MOD &100
170     STA base
180     LDA #location DIV &100
190     STA base + 1
200     LDY #0
210     LDA (base),Y      \Find length of list
220     STA len           \Store in len
230     LDX #0           \X will be zero on every pass except
                        the first
240
250     .repeat
260     STX flag          \Turn off swap flag
270     DEC len          \Reduce length by 1
280     LDX len
290     LDY #1
300
310     .nextpass
320     LDA (base),Y      \Fetch the next item
330     INY
340     CMP (base),Y      \Compare to following item
350     BCC swapdone      \If in right order, skip swap
360     BEQ swapdone
370     STA temp          \Store 1st item
380     LDA (base),Y      \Put 2nd item into A
390     DEY
```



```

400 STA (base),Y      \Put 2nd item in location of 1st
410 INY
420 LDA temp          \Get back 1st item
430 STA (base),Y      \Put 1st item in location of 2nd
440 DEC flag           \Set swap flag on
450
460 .swapdone
470 DEX
480 BNE nextpass
490 LDA flag           \Get swap flag
500 BNE repeat         \If yes, repeat search
510 RTS
520 J
530 NEXT pass
540
550 REM Test routine
560 REM Seed random number generator to produce fixed random
    sequence
570 X = RND(-2)
580 FOR I = &3001 TO &30FF
590   ?I = RND(&FF)
600 NEXT
610 ?&3000 = &FF
620
630 TIME = 0
640 CALL sort
650 time = TIME
660 PRINT "Time for sort was " ; time / 100 ; " seconds"
670 PRINT "Sorted list:"
680 FOR I = &3001 TO &30FF
690   PRINT TAB(((I - &3001) MOD 18) * 4) ; ?I ; " " ;
700 NEXT
710 PRINT

```

Program 6.3

The only tricky part of the program is the compare and swap procedure. If the items are in the right order, then

```

320 LDA (base),Y
330 INY
340 CMP (base),Y
350 BCC swapdone
360 BEQ swapdone

```

will have compared an item at $Y + 1$ with an item at Y . The exit through one of the branches will leave Y incremented ready for the next comparison. On the other hand, if the items are in the wrong order, lines 370 to 430 perform the swap. In the process Y is decremented (line 390) and then incremented (line 410) so the swap exit at line 440 still leaves Y ready incremented for the next comparison.

Test run

>RUN

Time for sort was 0.67 seconds

Sorted list:

```
1  1  1  2  2  2  3  4  4  5  6  7  9  9  9 10 10 10
11 12 14 14 14 14 16 18 20 21 22 22 22 23 24 24 24 25
25 25 27 28 28 28 30 30 32 35 36 38 38 38 39 39 40 42
43 44 44 46 47 51 52 56 59 61 61 61 63 63 63 63 63 64
64 66 66 67 71 71 72 72 72 73 73 77 78 80 80 80 81 82
82 82 85 85 86 93 93 94 95 96 96 97 98 98 99 99 99 100
102 103 106 107 107 107 107 107 109 109 110 112 113 114 114 118 118 119
119 119 121 121 122 122 122 123 124 125 126 126 127 127 129 129 129 133
134 135 137 137 138 138 138 138 139 143 143 147 149 149 151 152 154 154
158 158 158 161 162 162 163 163 165 169 170 171 176 178 182 183 183 185
186 187 190 190 190 190 192 192 193 193 193 193 195 196 200 200 200 200
200 200 201 201 201 203 205 207 208 212 212 213 213 213 215 215 216 217
217 218 219 219 220 221 223 223 224 225 225 226 229 229 229 232 232 233
234 236 236 237 237 239 239 244 244 245 245 246 246 247 253 254 254 255
255 255 255
```

K Load Program 6.3 and try it on blocks of numbers of your own.

(If you wish to sort an ASCII list using 0 as a delimiter, the length counter will no longer be needed. This means that, not only do you avoid one comparison in every loop, but, you can also then use the X register for 'flag' or 'temp' and produce a faster sort).

This may be the first program which we have presented that does something visibly faster than the BASIC programs which you will have previously met. As you have observed, Program 6.3 sorts 255 numbers in a fraction of a second. Now compare how long BASIC takes with Program 6.4 which includes a check on the time of the sort using TIME.

```
10 REM BASIC BUBBLE SORT
20
30 MODE 3
40 DIM XX(255)
50 REM Seed random number generator to produce same sequence
   as for Program 3
60 X = RND(-2)
70
80 FOR NX = 1 TO 255
90   XX(NX) = RND(255)
100  NEXT
110
120 TIME = 0
130
140 FOR KX = 1 TO 254
150   exchange = FALSE
160   FOR LX = KX + 1 TO 255
170     IF XX(LX) < XX(KX) THEN T% = XX(LX) : XX(LX) = XX(KX)
       : XX(KX) = T% : exchange = TRUE
180   NEXT LX
190   IF exchange THEN NEXT KX
```

```

200
210 now = TIME
220 PRINT "Sort time was " ; now / 100 ; " seconds." ''
230 FOR NX = 1 TO 255
240   PRINT TAB(((NX - 1) MOD 18) * 4) ; XX(NX) ; " " ;
250   NEXT

```

Test run

```

>RUN
Sort time was 111.12 seconds.

```

```

1  1  1  2  2  2  3  4  4  5  6  7  9  9  9  10 10 10
11 12 14 14 14 14 16 18 20 21 22 22 22 23 24 24 25
25 25 27 28 28 28 30 30 32 35 36 38 38 38 39 39 40
43 44 44 46 47 51 52 56 59 61 61 61 63 63 63 63 64
64 66 66 67 71 71 72 72 72 73 73 77 78 80 80 81 82
82 82 85 85 86 93 93 94 95 96 96 97 98 98 99 99 100
102 103 106 107 107 107 107 107 109 109 110 112 113 114 114 118 119
119 119 121 121 122 122 122 123 124 125 126 126 127 127 129 129 133
134 135 137 137 138 138 138 138 139 143 143 147 149 149 151 152 154
158 158 158 161 162 162 163 163 165 169 170 171 176 178 182 183 185
186 187 190 190 190 190 192 192 193 193 193 193 195 196 200 200 200
200 200 201 201 201 203 205 207 208 212 212 213 213 213 215 215 216
217 218 219 219 220 221 223 223 224 225 225 226 229 229 229 232 232
233 234 236 236 237 237 239 239 244 244 245 245 246 246 247 253 254
255 255 255

```

Program 6.4

BASIC, as you can see, here took 166 times as long as our machine-code program to do this sort. (However, the BASIC routine could have managed 4-byte integers which our machine-code program could not.)

SAQ 4

How would you modify Program 6.3 to sort in descending order?

6.5 Searching an ordered list

If we want to find whether or not an item is in a list, we can start at the beginning and work through, comparing each item against the search item. If the list is unordered then a sequential search of this type is the only method of searching.

Ordered lists can be searched more efficiently (unless they are very short). One such method for ordered lists is the 'bisection' (or 'binary') search. The method is as follows:

- ☐ Divide the list of items into half and ask 'is the target item equal to the half-way item, above it, or below it?'
- ☐ If 'equal' then the target item is in the list.
- ☐ If it is below, then discard the top half of the list (including the old middle item).

- ☐ If it is above, discard the bottom half (including the old middle item).
- ☐ Repeat the above procedure until the search item is found or there is no more list to search.

To keep track of what is happening, we label the end positions of the list 'low' and 'high'. These values change as we narrow down the section of the list to be searched.

On average, it will require $N/2$ comparisons to search an unsorted list of N items. A list of 1000 items will take 100 times as long to search as a list of 10 items. On the other hand, a binary search of list of 1000 sorted items takes, on average, only 3 to 4 times as long as a search of ten items.

Example 3

An ordered list contains the items A, F, I, M, P, T, U, Z. Use the bisection search procedure to find whether or not P is in the list. (In practice you shouldn't use a bisection search to search such a short list – a sequential search would be quicker.)

	1	2	3	4	5	6	7	8
	A	F	I	M	P	T	U	Z
low = 1: high = 8						4		
mid = $(1 + 8) / 2 = 4$ (ignore remainder)						M		
Divide list at item 4								
Does P = item 4? No								
Is P < item 4? No								
Discard bottom half and mid-point. So new low = 5.								
low = 5: high = 8							6	
mid = $(5 + 8) / 2 = 6$ (ignore remainder)							T	
Divide list at item 6								
Does P = item 6? No								
Is P < item 6? Yes								
Discard top half and mid point. So new high = 5.								
low = 5: high = 5							5	
mid = $(5 + 5) / 2 = 5$							P	
Divide list at item 5.								
Does P = item 5? Yes								
So P is in the list.								

SAQ 5

Use the above method to search for V. What happens?

Now that we have established how the 'not found' state is identified, we can draw the flowchart. This is shown in Figure 6.5.

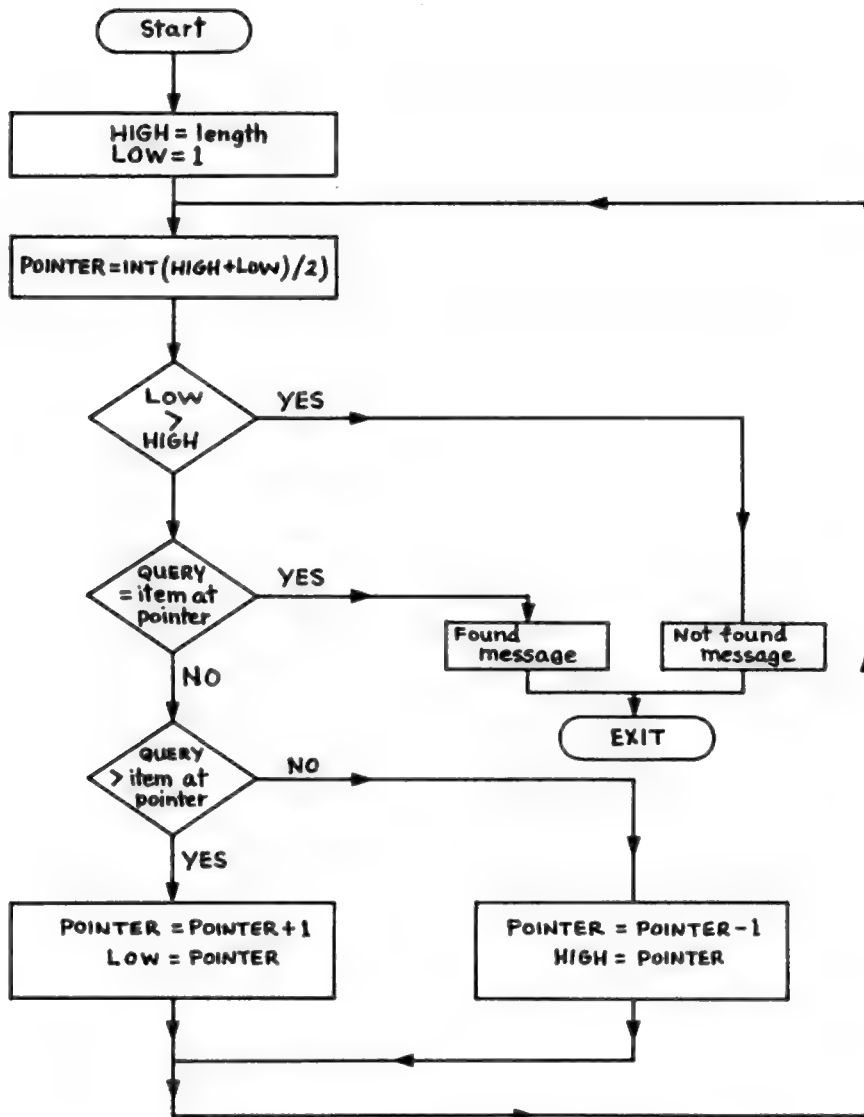


Figure 6.5 Flowchart for bisection search

The program memory usage

base	Contains the low byte of the address of the list	&70
	The high byte is in the following location	&71
low	The lower end of the part of the list being searched	&72

high	The upper end of the part of the list being searched	&73
query	The item being searched for	&74
carry } flag }	This is set to 1 if query is present and to 0 if query is not present	
result	At the end of the run, result holds the outcome of the search. If query was found, result holds its position in the list. If query was not found, result holds the position after which query would be if it were in the list.	&75

From line 270 onwards, the pointer to the current search position in the list is the Y register. (Prior to that, Y is used to set up the initial values of high and low.)

If you were going to use the program as a subroutine in a larger program then you would need a flag to indicate 'found' or 'not found'. We have omitted this from the program since we wanted a clear visual means of displaying 'found' or 'not found'. The program outputs a * if the query item is found and a ! if it is not found.

```

10 REM SEARCH
20
30 MODE 7
40
50 oswrch = &FFEE
60 star  = ASC "*"
70 pling = ASC "!"
80 base  = &70
90 low   = &72
100 high  = &73
110 query = &74
120 result = &75
130
140 DIM ZX100
150 FOR pass = 0 TO 3 STEP 3
160   PX = ZX
170   [
180   OPT pass
190
200   .search
210   LDY #0           \Pointer to base of list
220   LDA (base),Y     \Get length of list
230   STA high         \Store in 'high'
240   INY
250   STY low          \Set 'low' to 1

```

```

260
270 .loop
280 LDA high
290 CMP low      \Is high >= low?
300 BCC notfound \If not, query is not in list
310 CLC
320 ADC low      \Add low to high
330 ROR A        \Divide by 2 (discarding remainder)
340 TAY
350 LDA query    \Fetch search item
360 CMP (base),Y \Compare it to item at pointer
370 BCC lower    \Look lower down list
380 BEQ found    \Query is in list
390
400 .higher
410 INY          \Move middle up 1 item
420 STY low      \Make this the new low
430 BNE loop    \Y cannot be 0
440
450 .lower
460 DEY          \Move middle down 1 item
470 STY high     \Make this the new high
480 JMP loop
490
500 .found
510 STY result    \Found so, store position of item
520 LDA #star     \Load 'found' marker
530 JSR oswrch    \Output marker
540 RTS
550
560 .notfound
570 LDA high      \Item not found above high
580 STA result    \Store high as pointer to where item
                    belongs
590 LDA #pling    \Load 'not found' marker
600 JSR oswrch    \Output marker
610
620 RTS
630 ]
640 NEXT pass
650
660 REM Test routine
670 REM Put test list into memory
680 ?&3001 = 42
690 ?&3002 = 56
700 ?&3003 = 67
710 ?&3004 = 72
720 ?&3005 = 80

```

```

730 ?&3006 = 90
740 REM Put length at base
750 ?&3000 = 6
760 ?base = 0
770 base?1 = &30
780
790 REPEAT
800 PRINT
810 INPUT "Enter search item " target
820 ?query = target
830 PRINT "Result (* = in list, ! = not in list): " ;
840 CALL search
850 PRINT " " "Position of item/position after which item
    would be: " ; ?result
860 UNTIL FALSE

```

Program 6.5

Test run

>RUN

```

Enter search item 67
Result (* = in list, ! = not in list): *
Position of item/position after which item would be: 3

```

```

Enter search item 95
Result (* = in list, ! = not in list): !
Position of item/position after which item would be: 6

```

6.6 Inserting an item into an ordered list

We wrote the bisection search routine in the last section so that it provides us with the position that 'query' would occupy if it were in the list. (Actually it gives the position after which it would appear). This is essential information if we are to be able to insert a new item into an ordered list. (We are assuming that we are only dealing with lists in which no two items are the same so that a new item will not be added to the list if it is already there).

Suppose we have a list of length L and wish to insert a new item after item P . The procedure for doing this is shown in Figure 6.6.

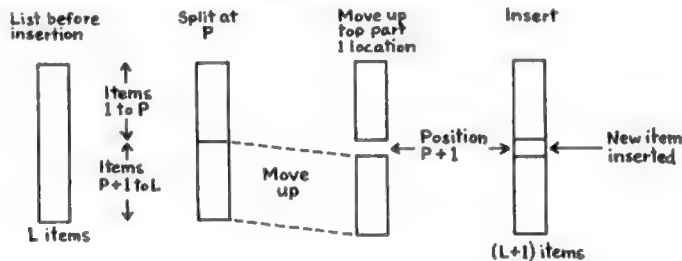


Figure 6.6 Adding an item to an ordered list of length L

SAQ 6

Draw an outline flowchart for inserting an item into an ordered list of length L. The new item is to go in after item P. Include in the flowchart a check on whether or not the query item is in the list.

We can now turn this into an assembly language program. The program will first call 'search' (Program 6.5) as a subroutine. If the query item is in the list, that is the end of the run. If it is not in the list, 'search' tells us where to insert the item. We subtract 'result' from 'length' (lines 150-180) and store the value in the X register. X holds the number of items to move up in order to create the space for the new item.

190 - 250	Find whether or not 'query' is already in the list.
270 - 350	Find how many items need to be moved.
370 - 440	Move the items up to create the space.
460 - 500	Insert the new item ('query').
510 - 530	Store the new length back at the base of the list.

Calling the 'search' subroutine

This is the first program in which we need to incorporate a previous program as a subroutine. This is done in a similar manner to calling a subroutine in BASIC. First, note that the 'search' program (like all assembly programs) is of the form

Label	.search
	...
	...
	...
Return instruction	RTS

From BASIC we would call such a program by a label (CALL search) or memory location (CALL Z%). But if we call it by

JSR search

from within our program, the search subroutine will be executed and then the program will continue with the next instruction after JSR search.

Here is the full insert program incorporating 'search' as a subroutine.

```
10 REM INSERT
20
30 MODE 7
40 listloclo = 0
50 listlochi = &30
60 base = &70 : REM Plus &71
70 low = &72
80 high = &73
90 query = &74
100 result = &75
110 length = &76
120
130 DIM Z%150
140 FOR pass = 0 TO 3 STEP 3
150   P% = Z%
160   [
170     OPT pass
180
190     .insert
200     LDA #listloclo
210     STA base
220     LDA #listlochi
230     STA base + 1
240     JSR search      \Find if query is already in list
250     BCS exit        \If in list, skip insert routine
260
270     .additem
280     LDY #0          \Set pointer to base of list
290     LDA (base),Y
300     STA length      \Find length of list
310     TAY
320     SEC
330     SBC result       \Find No of items above query
340     BEQ itemin       \If query goes at top of list, skip move
350     TAX              \Store in X
360
370     .move
380     LDA (base),Y     \Get item to move up
390     INY
400     STA (base),Y     \Store it one location up
410     DEY              \Move pointer back for next item
420     DEY              \Move pointer back for next item
430     DEX              \Decrement item counter
440     BNE move         \Loop back if more to move
450
460     .itemin
470     INY              \Move pointer up
```

480	LDA query	\Get query item
490	STA (base),Y	\Insert in gap
500	INC length	\Add 1 to length
510	LDA length	
520	LDY #0	\Pointer to base of list
530	STA (base),Y	\Store new length
540		
550	.exit	
560	RTS	
570		
580	.search	
590	LDA #0	
600	TAY	\Pointer to base of list
610	LDA (base),Y	\Find length of list
620	STA high	\Store in 'high'
630	LDA #1	
640	STA low	\Set 'low' to 1
650		
660	.checkend	
670	LDA high	
680	CMP low	\Is high >= low?
690	BCC notfound	\If not, item not in list
700	CLC	
710	ADC low	\Add low to high
720	ROR A	\Divide by 2
730	TAY	
740	LDA query	\Fetch search item
750	CMP (base),Y	\Compare with item at pointer
760	BCC lower	\Look lower down list
770	BEQ found	\Item is in list
780		
790	.higher	
800	INY	\Move mid up 1 item
810	STY low	\Make this the new low
820	JMP checkend	
830		
840	.lower	
850	DEY	\Move mid down 1 item
860	STY high	\Make this the new high
870	JMP checkend	
880		
890	.found	
900	TYA	\Found, so
910	STA result	\Store position of item
920	SEC	\And set carry
930	JMP searchdone	
940		
950	.notfound	
960	LDA high	\Item not found above high
970	STA result	\Store high as pointer to where item belongs
980	CLC	\Item not found. Clear carry

```

990
1000 .searchdone
1010 RTS
1020 ]
1030 NEXT pass
1040 REM Test routine
1050 REM Put list into memory at &30000
1060 ?&3001 = 50
1070 ?&3002 = 60
1080 ?&3003 = 70
1090 REM Store length of list
1100 ?&3000 = 3
1110
1120 REPEAT
1130 PRINT "List is now as follows:"
1140 FOR I = &3001 TO &3001 + ?&3000 - 1
1150 PRINT ; I ; " " ;
1160 NEXT
1170 PRINT '
1180 INPUT "Enter item to insert in list: " query
1190 ?&74 = query
1200 CALL insert
1210 UNTIL FALSE

```

Program 6.6

Test run

```

>RUN
List is now as follows:
50 60 70

Enter item to insert in list: 45
List is now as follows:
45 50 60 70

Enter item to insert in list: 67
List is now as follows:
45 50 60 67 70

Enter item to insert in list: 60
List is now as follows:
45 50 60 67 70

Enter item to insert in list: 0
List is now as follows:
0 45 50 60 67 70

```

Program 6.6

SAQ 7

The 'insert' program does not check that there is room in the list for the new item. How would you modify the program to incorporate a check?

Extending the insert program

If you wanted a full insert and delete routine then you could extend Program 6.6 to do this, using the memory block move routine.

6.7 Putting lists in memory

You have seen that creating strings or lists of data in assembly language is tedious. One way of avoiding this is to place the data as a string at a fixed memory location using BASIC. This string can then be accessed using assembly language. In this way we can place lists and tables in memory using BASIC (which is slow) but use them using assembly language (which is fast).

Placing a string of letters in memory

As you have previously seen, the statement

```
DIM S%20
```

will reserve 20 bytes of memory at the location 'pointed to' by S%. If it is followed by the statement

```
$$% = "TEST STRING"
```

then the string "TEST STRING" will be in bytes 0 to 10 of the 20 bytes with &D (carriage return) in byte 11. Notice that the \$ sign precedes S% (\$S% is not the same as \$\$). You can access any one byte of the string by

```
S%?N
```

where N is a number indicating the position required in the string.

This is demonstrated by Program 6.7 which:

```
40          Puts "TEST STRING" into memory at $$%
80          Prints out its 4th character using S%?3. (The
           first character is S%?0.)
120 - 140   Prints out the whole string character by
           character using S%?N

10 REM FIXED LOCATION STRINGS
20
30 DIM S%20
40 $$% = "TEST STRING"
50
60 REM Access a single character
70 PRINT "4th character of string: " ;
80 PRINT CHR$(S%?3)
90
100 REM Or access the whole of the string
110 PRINT "Full string at S%: " ;
120 FOR N = 0 TO LEN($$%) - 1
130   PRINT CHR$(S%?N) ;
140   NEXT
150 PRINT
```

Program 6.7

Test run

```
>RUN
4th character of string: T
Full string at S$: TEST STRING
```

You can also change any one byte of the string with the statement:

```
S$?N = ...
```

In Program 6.8, you can change any single character in the string S\$. Notice that the character input into C\$ (e.g. B) must be changed into its ASCII code (66 for B) before being placed in the string. This is done with `ASC(C$)`.

```
10 REM CHARACTER CHANGE IN FIXED STRINGS
20
30 DIM S$20
40 S$ = "TEST STRING"
50
60 INPUT "Position to be changed " P
70 INPUT "New character " C$
80 REM Put new character at position P in S$
90 S$?P = ASC(C$)
100 PRINT S$
110
120 PRINT "Contents of full string at S$: " ;
130 FOR N = 0 TO LEN(S$) - 1
140   PRINT CHR$(S$?N) ;
150 NEXT
160 PRINT
```

Program 6.8

Test run

```
>RUN
Position to be changed 3
New character B
TESB STRING
Contents of full string at S$: TESB STRING
```

This method can also be used to put single byte numbers into a list in memory. If we save space for a list L of 20 bytes using `DIM L$20` we can then place single byte numbers into that list. For example:

```
>DIM L$20
>L$?2 = &08           Puts &08 into byte 2.
>L$?6 = &2A           Puts &2A into byte 6.
```

Check:

```
>P.~LX?2
      8
>P.~LX?6
     2A
```

We can adapt the technique by using ! (pling) in place of ? to allow us to place 4 one-byte numbers into consecutive locations.

Example 4

Store the numbers

```
&5, &3, &57, &49, &82, &17, &A3, &1B
```

in positions 0 to 8 in a list S%.

Solution

We could do this byte by byte using S%?N = ... but it only needs two statements using !:

```
S%!0 = &49570305
S%!4 = &1BA31782
```

To see how this works, let's examine the second of these two statements.

insert 4 bytes starting at position 4 in S%

```
S%!4 = &1BA31782
      ^      ^
      msb   lsb
```

If we were to examine the memory after executing S%!0 = &49570305 and S%!4 = &1BA31782 we would find:

S%		5	3	57	49	82	17	A3	1B	
Byte No.	0	1	2	3	4	5	6	7	8	...

You can also place large decimal numbers into memory using !.

Example 5

```
!&3000 = 12345678
```

places the number 12345678_{10} into the four memory locations &3000 to &3003 where it is stored as

```
78 97 188 0.
```

You can confirm that it is 12345678 by calculating

```
78 + (97 * 256) + (188 * 2562) + (0 * 2563) = 12345678.
```

Or, if you want to know the value of the four byte number stored at &3000 to &3003, type

```
P. !&3000.
```

The result, 12345678, is printed out.

SAQ 8

Write a BASIC program to place a string of your choice at a fixed location. Then replace all A's and a's in the string with *. Make the program print out the revised string from memory to confirm the changes.

SAQ 9

Write a BASIC program to place ten numbers in memory. Then search the list to see whether the number 65 is present. If it is, change it to 97. Make the program print out the revised list. (The numbers must be less than &100).

Using \$space as an address

When you reserve memory with a statement such as

```
DIM $space%50
```

the variable `space%` takes the value of the address where the computer has placed `$space%`. You can use `space%` as an address in assembly language programs and write instructions such as

```
LDA space%, 'Y'
```

This we shall now do in some examples.

6.8 Look-up tables

We looked at the last section in terms of BASIC but it easily extends to assembly language. If the lists are placed in memory using BASIC, they can be accessed using assembly language. For example, to access the 3rd number (byte 2) of the list `S%`, we write

```
LDY #2  
LDA S%,Y
```

or,

```
LDX #2  
LDA S%,X
```

This will load the accumulator with the 3rd number in the list `S%`. We shall use this to write a program to code words and phrases. The method is called a 'look-up table' since we look up a list in memory to see what is at a given position.

Simple code program

The code works by listing the alphabet plus the character 'space' in the usual order. Then the same characters are listed in any order you like underneath. To code a word, find each character in the normal order

list and replace it with whichever character is in the other list. For our demonstration program, the code is very simple:

Alphabet	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z #
Code	# Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

where # indicates a space between words.

e.g. HELLO codes as TWPPM

HELLO FRED codes as TWPPMAVJWX

A flowchart for the coding process is shown in Figure 6.7.

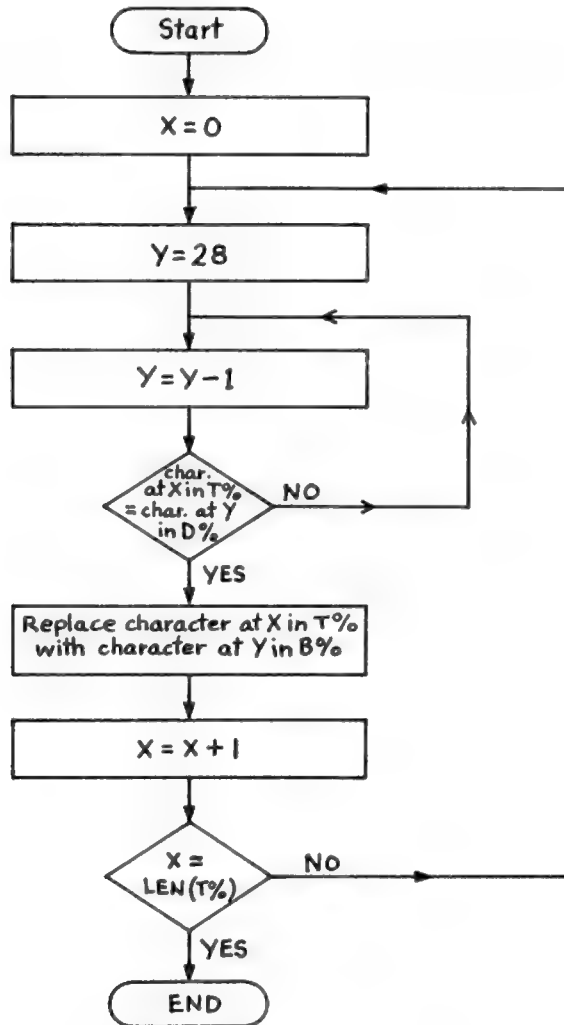


Figure 6.7 Simple code routine

The program for the simple code is shown in Program 6.9.

```
10 REM SIMPLE CODE 1
20 DIM D%27, B%27, T%30, Z%100
30 $D% = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
40 $B% = "ZYXWVUTSRQPONMLKJIHGFEDCBA"
50 oswrch = &FFEE
60
70 FOR pass = 0 TO 3 STEP 3
80   P% = Z%
90   [
100    OPT pass
110
120    .code
130    LDX #0          \Set character position counter
140
150    .next
160    LDY #28         \Alphabet pointer - works from top to bottom
170
180    .search
190    DEY             \Move alphabet pointer to next character
200    LDA T%,X        \Fetch the next character from string
210    CMP D%,Y        \Same as alphabet character at Y?
220    BNE search      \If no, loop back
230    LDA B%,Y        \If yes, get code character from B%
240    STA T%,X        \And substitute in string for alphabet
                        character
250    JSR oswrch       \Print code character just inserted
260    INX              \Move along to next character to code
270    CMP #&D         \Was last character 'return'?
280    BNE next        \If no, loop back
290
300    RTS
310  ]
320  NEXT pass
330
340 REM Test routine
350 REPEAT
360   INPUT "Enter a string to be coded: " $T%
370   PRINT "Coded string is: ";
380   CALL code
390   PRINT '
400   UNTIL FALSE
```

Program 6.9

Test run

```
>RUN
Enter a string to be coded: TEST STRING
Coded string is: HWIHAIHJSNU

Enter a string to be coded: THIS IS A CODING PROGRAM
Coded string is: HTSIASIA AYMXSNUALJMUJ O
```

Program 6.9 is not a very efficient method of doing this coding operation – it can be done much more easily without using look-up tables since we can convert ASCII letter codes 65 to 90 to 1 to 26 using masking (see section 7.9). The more efficient program would then be:

```
10 REM SIMPLE CODE 3
20 DIM BX27, TX30, ZX100
30 $B% = " ZYXWVUTSRQPONMLKJINGFEDCBA"
40 oswrch = &FFEE
50
60 FOR pass = 0 TO 3 STEP 3
70   P% = ZX%
80   [
90     OPT pass
100
110    .code
120    LDX #0      \Set character position counter
130
140    .next
150    LDA TX,X    \Fetch the next character from the string
160    CMP #&D
170    BEQ exit    \Exit if character is 'return'
180    AND #&1F    \Convert ASCII letter to code between 1 and 26
190    TAY         \Position of letter now becomes pointer
200    LDA B%,Y    \Point to letter in code list
210    JSR oswrch  \Print converted letter
220    INX
230    BNE next    \Loop back
240
250    .exit
260    RTS
270  ]
280  NEXT pass
290
300 REM Test routine
310 REPEAT
320   INPUT "Enter a string to be coded: " $TX
330   PRINT "Coded string is: " ;
340   CALL code
350   PRINT '
360   UNTIL FALSE
```

Program 6.9b

Conversion program

An assembly language program may require data which has to be converted e.g. inches to centimetres or degrees Fahrenheit to degrees Centigrade. This could be done by coding the conversion formula but it is often easier to use a look-up table which stores all the possible conversion results. It is also faster since the look-up process is extremely quick. Here is a program to convert a temperature in Fahrenheit to Centigrade by means of a look-up table.

First we must store the temperatures in a list. This is done in BASIC before the program is assembled:

```
10 DIM B%212
20 FOR F = 32 TO 212
30   B%?F = (5 / 9) * (F - 32)
40   NEXT
```

The list B% can be accessed by the assembly language routine at extremely high speed. The full program is shown in Program 6.10.

```
10 REM TEMPERATURE LOOK-UP
20
30 DIM B%212, Z%50
40
50 REM Set up look-up table
60 FOR F = 32 TO 212
70   B%?F = (5 / 9) * (F - 32)
80   NEXT
90 Faren = &70
100 Cent = &71
110 P% = Z%
120 [
130 OPT 1
140
150 .convert
160 LDY Faren    Fetch temp in Fahrenheit
170 LDA B%,Y     Look up temp in Centigrade
180 STA Cent     Store result in Cent
190
200 RTS
210 ]
220
230 REM Test routine
240
250 REPEAT
260
270   INPUT "Enter temperature in Fahrenheit: " F
   ?&70 = F
290   CALL convert
300   PRINT "Temperature in Centigrade is: " ; ?&71
310
320   UNTIL FALSE
```

Program 6.10

Test run

```
>RUN
Enter temperature in Fahrenheit: 32
Temperature in Centigrade is: 0
Enter temperature in Fahrenheit: 212
Temperature in Centigrade is: 100
Enter temperature in Fahrenheit: 89
Temperature in Centigrade is: 31
```

Program 6.10

You will notice that at the start of the run of Program 6.10 there is a pause whilst the look-up table is filled by the BASIC program at lines 60-80. It is important to fill look-up tables before they are needed. In this way they can be accessed at very high speed.

If you need more than one look-up table in a program then it is best to use indirect addressing. You can then switch from one table to another just by changing the base address pointer. Additionally, with indirect addressing it becomes possible to have a table of more than &100.

Assignment F

1. Modify Program 6.6 to make it delete an item from a list. Test your program on a suitable list.
2. Modify Program 6.3 to cater for 2-bit items.

Objectives of Unit 6

In the objectives below, all lists refer to lists of 1-byte numbers.

After studying this unit you should be able to write programs to:

- ☐ Use OSRDCH to input characters from the keyboard.
- ☐ Set up a list in memory starting at a base address and incorporating a record of its own length.
- ☐ Print out a list from memory.
- ☐ Sort a list in memory.
- ☐ Search an ordered list to find whether or not an item is in it. If the item is not in the list, the program should indicate where the item should be in the list.
- ☐ Insert an item into an ordered list.
- ☐ Use look-up tables.

You should also be able to use the following techniques:

- ☐ \$ as a prefix to a string to place it in memory space reserved for the string.
- ☐ ?N to access the Nth byte of a string.
- ☐ ?N to alter the Nth byte of a string.
- ☐ ! to access a byte of a 4-byte number.

Answers to SAQs

SAQ 1

```
10 REM INPUT
20
30 MODE 7
40 listloclo = &0
50 listlochi = &30
60 base = &70 : REM Plus &71
70 eol = &0D
80 osrdch = &FFE0
90 oswrch = &FFEE
100
110 DIM ZX100
120 FOR pass = 0 TO 3 STEP 3
130   PX = ZX
140   [
150     OPT pass
160
170     .input
180     LDA #listloclo \Get low-byte of address to store
                        \list
190     STA &70         \Store in zero Page
200     LDA #listlochi \Get high-byte of address to store
                        \list
210     STA &71         \Store in zero Page
220
230     LDY #0         \Initialise character counter
240
250     .next
260     JSR osrdch      \Get next character
270     JSR oswrch      \Print character on screen
280     STA (base),Y    \Store character in memory
290     INY             \Increment character counter
300     CMP #eol        \Was last character 'return'?
310     BNE next        \If not, loop back for next character
320     LDA #0          \Set 0 for end of list
330     STA (base),Y    \Store 0 at end of list
340
350     RTS
360   ]
370 NEXT pass
380
390 REM Test routine
400 PRINT "Enter characters for input routine."
410 CALL input
420
430 REM Print out the contents of the memory block
440 PRINT "Characters read back from memory block:"
450 N = &3000 - 1
460 REPEAT
470   N = N + 1
480   IF ?N <> 0 THEN PRINT CHR$(?N) ;
490 UNTIL ?N = 0
500 PRINT
```

Test run

>RUN

Characters input at keyboard:

THIS SENTENCE WAS PUT INTO MEMORY AT &3000

Print out the contents of the memory block:

THIS SENTENCE WAS PUT INTO MEMORY AT &3000

The BBC Micro has a much more powerful input routine that can accept a whole line of input. This routine, OSWORD with A = 0, will be described in Unit 8.

SAQ 2

```
10 REM INPUT
20
30 osrdch = &FFEO
40 oswrch = &FFEE
50 DIM Z%100
60
70 FOR pass = 0 TO 3 STEP 3
80   P% = Z%
90   [
100  OPT pass
110
120  .input
130  LDX #1          \Set character counter
140
150  .next
160  JSR osrdch      \Get next character
170  JSR oswrch      \Print on screen
180  CMP #&0D       \Is it return?
190  BEQ length      \If yes, branch and store length
200  STA &3000,X     \Store character in memory
210  INX             \Add one to counter for next
                    \character
220  BNE next        \Loop back for next character
230
240  .length
250  DEX             \Adjust counter to length
260  STX &3000       \Store length at start of list
```

```

270
280   RTS
290 ]
300   NEXT pass
310
320 REM Test routine
330 REPEAT
340   PRINT "Enter a string: ";
350
360   CALL input
370
380   PRINT "Read back string from memory
      block: ";
390   length = &3000
400   IF length = 0 THEN 440
410   FOR I = &3001 TO &3000 + length
420     PRINT CHR$(?I) ;
430     NEXT
440   PRINT
450   UNTIL FALSE

```

Program 6.12

Test run

>RUN

Enter a string: TEST STRING

Read back string from memory block: TEST STRING

Enter a string:

Read back string from memory block:

Enter a string: A

Read back string from memory block: A

(Note that we have tested that the routine will cope with a null entry).
The differences between the two programs are:

- (a) The base address (&3000) is used in the program itself and is not stored in zero page.
- (b) X is now the counter.

SAQ 3

```
10 REM LIST PRINT
20 ?&70 = 0
30 ?&71 = &30
40 oswrch = &FFEE
50 osrdch = &FFED
60 space = ASC " "
70
80 DIM Z%50
90 FOR pass = 0 TO 3 STEP 3
100   P% = Z%
110   [
120     OPT pass
130
140     .print
150     LDY #0           \Pointer to base of list
160     LDA (&70),Y      \Load length into A
170     TAX
180
190     .next
200     INY              \Move pointer to next item
210     LDA (&70),Y      \Load item into A
220     JSR oswrch        \Print character
230     LDA #space       \Space character
240     JSR oswrch        \Print a space
250     DEX
260     BNE next          \End of list?
270
280     RTS
290   ]
300 NEXT pass
310
320 REM Test routine
330
340 CALL print
```

Program 6.13

Test run

```
>RUN
R O Y G B I V >
```

SAQ 4

Change line 350 to 'BCS swapdone' and delete 360.

☒ Alter Program 6.2 with this modification and confirm that it works.

SAQ 5

1	2	3	4	5	6	7	8
A	F	I	M	P	T	U	Z

low = 1: high = 8

Mid = $(1 + 8) / 2 = 4$

Divide list at item 4

4

Does V = item 4? No

M

Is V item 4? No

Discard bottom half and mid point. So new low = 5.

Mid = $(5 + 8) / 2 = 6$

6

Divide list at item 6.

T

Does V = item 6? No

Is V item 6? No

Discard bottom half and mid-point. So new low = 6.

Mid = $(6 + 8) / 2 = 7$

7

Divide list at item 7.

U

Does V = item 7? No

Is V item 7? No

Discard bottom half and mid-point. So new low = 7.

Mid = $(7 + 8) / 2 = 7$

7

Divide list at item 7.

U

Does V = item 7? No

Is V item 7? No

Discard bottom half and mid-point. So new low = 8.

Mid = $(8 + 8) / 2 = 8$

8

Divide list at item 8.

Z

Does V = item 8? No

Is V item 8? Yes

Discard top half and mid-point. So new high = 7.

Now low = 8 and high = 7, which doesn't make sense. This has occurred because V is not in the list. If the item is not in the list the search results in low exceeding high.

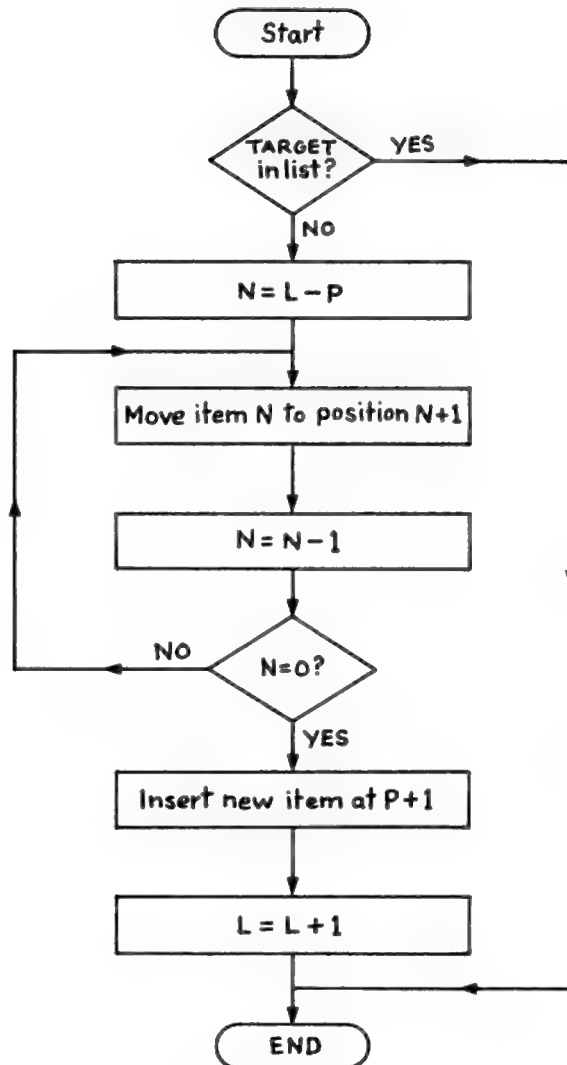


Figure 6.8 Inserting an item into an ordered list

SAQ 7

Add

```
294 CMP #&FF      \Compare to maximum length
296 BCS exit       \If full, leave subroutine
```

In addition, for a working program, you need to set a 'list full' indicator to warn the user that the item was not added.

SAQ 8

```
10 REM REPLACE ROUTINE
20
30 DIM S$30
40 REPEAT
50
60   INPUT "Enter string to be processed " S$
70   FOR IX = 0 TO LEN(S$)
80     IF S$?IX = ASC("A") OR S$?IX = ASC("a")
      THEN S$?IX = ASC("*")
90   NEXT IX
100  PRINT S$
110  PRINT
120
130  UNTIL FALSE
```

Program 6.14

Test run

```
>RUN
Enter string to be processed HELLO
HELLO

Enter string to be processed ART
*RT

Enter string to be processed a lot of aaaaaaaa...s
* lot of *****...s
```

SAQ 9

```
10 REM REPLACE ROUTINE
20
30 MODE 3
40 DIM H$10
50 FOR IX = 0 TO 9
60   INPUT "Next number? " number
70   H$?IX = number
80   NEXT IX
90 PRINT "Numbers as entered: " TAB(28) ; : PROCprintnumber
100 FOR IX = 0 TO 9
110   IF H$?IX = 65 THEN H$?IX = 97
120   NEXT IX
130 PRINT "Numbers after processing: " TAB(28) ; : PROCprintnumber
140 END
```

```

150
160 DEF PROCprintnumber
170 FOR K% = 0 TO 9
180   PRINT ; HX?K% ; " " ;
190 NEXT K%
200 PRINT
210 ENDPROC

```

Test run

```

>RUN
Next number? 12
Next number? 67
Next number? 89
Next number? 65
Next number? 34
Next number? 2
Next number? 8
Next number? 0
Next number? 12
Next number? 16
Numbers as entered:      12 67 89 65 34 2 8 0 12 16
Numbers after processing: 12 67 89 97 34 2 8 0 12 16

```

UNIT 7

The stack, CALL, USR and masking

- 7.1 How subroutines work
- 7.2 The stack
- 7.3 Using the stack as memory
- 7.4 Saving the registers on the stack
- 7.5 Passing parameters via the stack
- 7.6 Passing parameters via A%, X%, Y% and C%
- 7.7 Passing parameters via CALL
- 7.8 USR
- 7.9 AND, ORA and EOR

Assignment G

Objectives of Unit 7

Answers to SAQs

7.1 How subroutines work

All the programs we have written (except Program 6.6) have been written as single routines. Program 6.6 contained two subroutines (search and additem). search was called by JSR search after which the second routine, additem, was executed if the search showed that the query item was not in the list. We have also used some of the built-in operating system subroutines such as OSWRCH. We are now going to consider in more detail how a subroutine works.

To keep things simple, we shall look at a program that just displays two characters. In Program 7.1, the subroutine OSWRCH is called twice. Each time it is called, the program execution jumps to a machine code language routine which is executed before control is handed back to the main part of the program.

```
10 REM SUBROUTINES
20
30 oswrch = &FFEE
40
50 PX = &2000
60 [OPT 1
70
80 .display
90 LDA #ASC"A"
100 JSR oswrch
110 LDA #ASC"B"
120 JSR oswrch
130
140 RTS
150 ]
160
170 CALL display
```

Program 7.1

Execution of this program follows the pattern shown Figure 7.1.

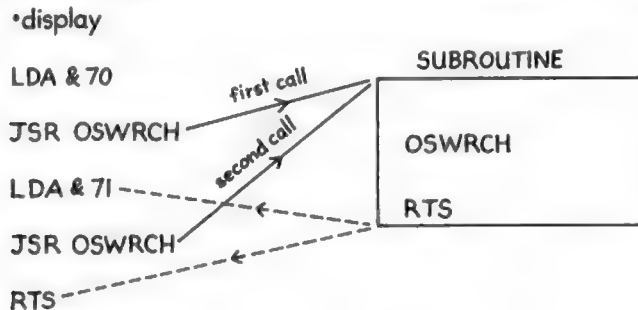


Figure 7.1 Two calls of the OSWRCH subroutine

The method by which the program calls the subroutine is clear enough since the call itself tells us. i.e.

`JSR oswrch` where `oswrch` is `&FFEE`

tells the program to go to memory location `&FFEE` and continue with whatever instructions it finds there. But how does the subroutine transfer control back to the program and how does the program know which instruction to do next? The answer lies in the *stack*.

7.2 The stack

The stack is an area of memory. In the case of the 6502 microprocessor, the stack area is Page One i.e. locations `&100` to `&1FF`. It is shown in Figure 7.2.

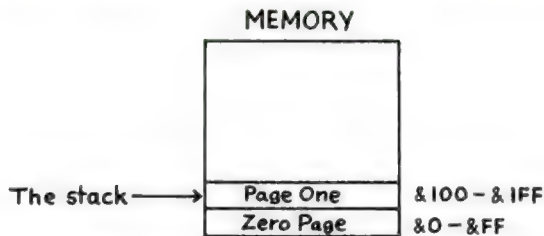


Figure 7.2 The stack resides in Page One

For most of the time you do not address the stack by memory location. Instead you can put a number *onto* the top of the stack or pull a number *off* the top of the stack. Hence the name *stack*. It behaves like a stack of plates. If you want a plate you take it from the top of the pile. If you want to add a plate to the pile, you put it on the top. The plate analogy is illustrated in Figure 7.3a. However, the stack differs from a pile of plates in that all `&100` locations in Page One form the stack and the locations themselves are not taken on and off the stack. Instead, a pointer called the *stack pointer* points to the current next free location in the stack and, just as with any other memory location, data is stored in, and copied out of, the stack (see Figure 7.3b).

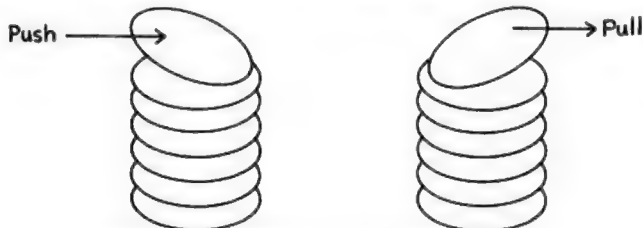


Figure 7.3a Pushing and pulling plates off a pile

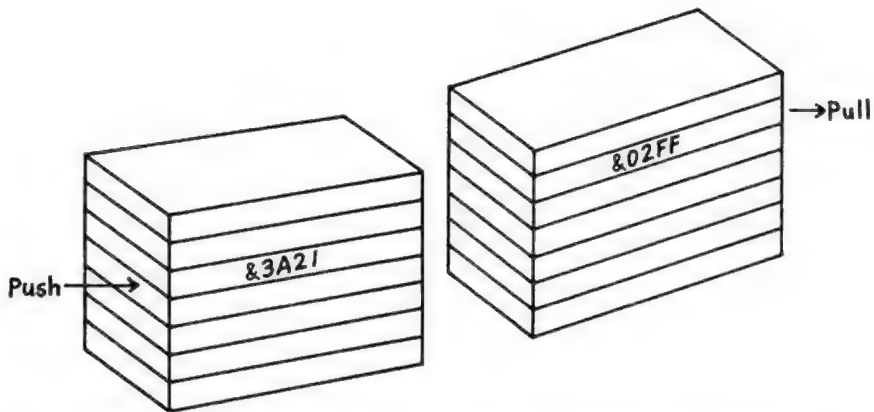


Figure 7.3b Pushing and pulling numbers from the Page One stack

The stack is said to have a LIFO structure: Last In First Out. In other words, if you put data onto the stack, the last item which you put in will always be the item on the top. So, when you come to take an item off the stack, it is bound to be the last one that you put in.

So far, we have emphasised accessing the stack through pushing and pulling and for that you do not need to know the address of the top of the stack. It is also possible to access any memory location in the stack using the stack pointer. However, it is dangerous to do so since the stack has the special job of keeping track of where the program is. If you make a mistake with your use of the stack pointer, you are likely to cause your program to crash. We therefore advise you to stick to using the stack commands PHA, PLA, PHP and PLP (which we will shortly introduce).

Use of the stack during subroutines

The stack has several uses. The first is the CPU's automatic use of the stack when you call a subroutine.

When a subroutine is called, the program counter contains (as usual) the address of the next instruction to be executed. But we want to divert the program to a subroutine and so prevent it from continuing straight on to the next instruction. In executing the JSR instruction, the microprocessor puts the contents of the program counter onto the top of the stack. This saves the address at which program execution will continue after the subroutine has been executed. It also, of course, frees the program counter for use by the subroutine. At the end of the subroutine, the statement RTS causes the microprocessor to fetch the original address from the top of the stack and put it back into the program counter.

So, in Program 7.1, at the first call of 'JSR oswrch', the program counter contains &2005 which is the address of LDA #ASC"B" – the instruction that will follow the subroutine call. It is this address that is saved on

the stack for the program to return to after the subroutine. This process is illustrated in Figure 7.4. Notice that we do not know what is on the top of the stack at the start of the subroutine call, and nor do we need to know.

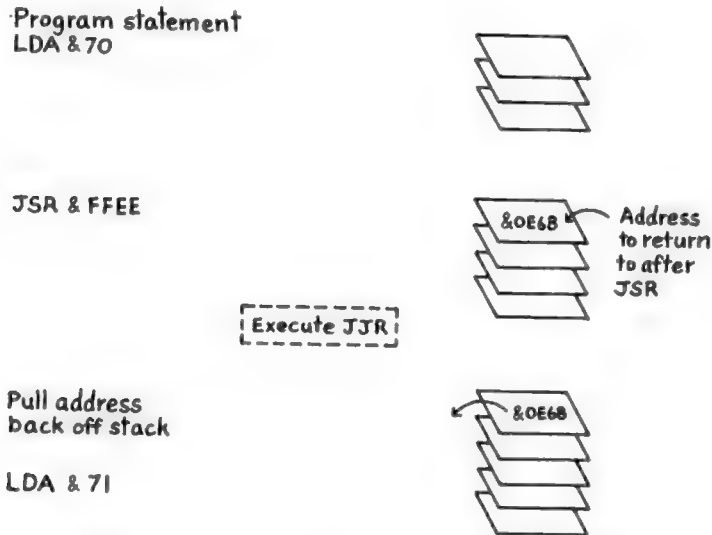


Figure 7.4 Use of the stack to store a return address

We have made one simplification in the above illustration. Program counter addresses are 16-bit numbers but the stack is an 8-bit memory block. The return address must be held on the stack as two 8-bit numbers. The high-byte goes on first, followed by the low-byte.

The stack pointer

You may wonder how the microprocessor knows where the top of the stack is. It does this through the **stack register** (see Figure 2.1) which contains the address of the next free location in the stack (unless deliberately changed by the program). Since this is an automatic feature of the CPU, you will not need to learn to use the stack pointer although stack pointer control commands are available in the 6502 instruction set for use in advanced programming.

SAQ 1

Draw an illustration to show the state of the top of the stack during the second call of OSWRCH in Program 7.1.

SAQ 2

Do we know what numbers will be in the program counter during execution of &FFEE?

Subroutines calling subroutines

There is nothing to stop a subroutine calling a subroutine which calls a subroutine ... (until the stack is full!). The stack still looks after the

proper order of program execution. Consider the case illustrated in Figure 7.5 where hypothetical subroutines at hypothetical addresses are called one after the other.

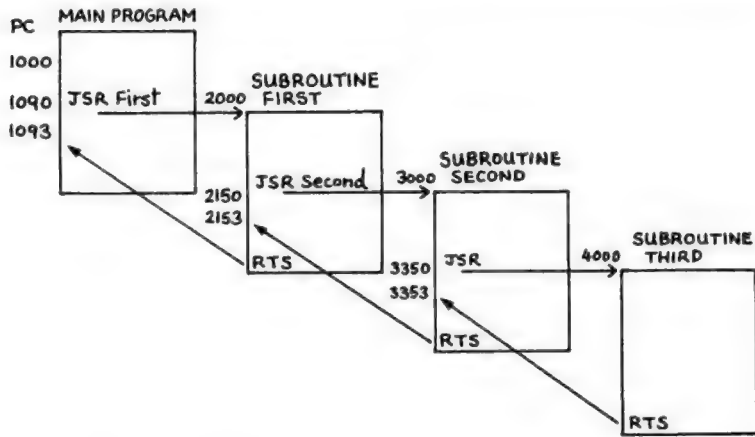


Figure 7.5 Subroutines calling subroutines

The stack (simplified to show 16-bit numbers placed on without splitting into high- and low-bytes) will appear as in Figure 7.6.

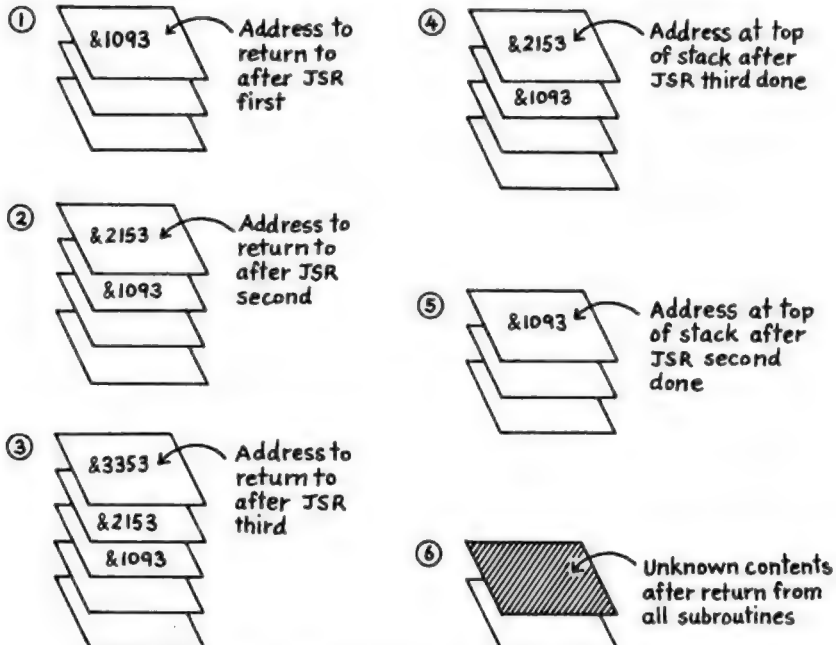


Figure 7.6 The stack during repeated subroutine calls

SAQ 3

The stack is in Page One which is 256 bytes long. What is the maximum number of nested subroutine calls that can be handled at any one time?

From this discussion you can now see that RTS is capable of a more precise definition than we gave it in Unit 2. For, whilst ReTurn from Subroutine is a partial explanation of its action, RTS is an instruction to take the address off the top of the stack and to put it into the program counter.

RTS Transfer address on top of stack to program counter

The stack is upside down

One other point about the 6502 stack is that it is 'upside down' i.e. when you switch on, the operating system arranges for the first free location to be 255 and the stack is filled from this location downwards towards 0. It rather spoils the analogy with the pile of plates, but apart from that it makes no difference to what we have said so far.

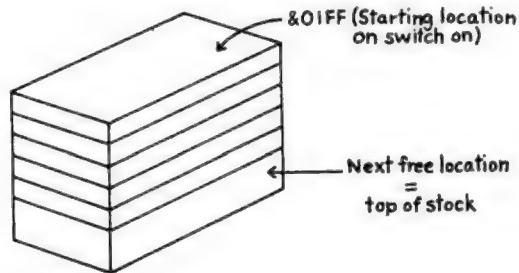


Figure 7.7 The stack is really upside down

Everything we have said so far concerns the CPU's automatic use of the stack to control program flow during subroutines. We shall now look at programmer-controlled uses of the stack.

7.3 Using the stack as memory

The stack can be useful as a temporary storage location area i.e. a place to put a few items of data knowing that they will have to be pulled off before whatever else is on the stack is next needed. Data can be transferred to and from the stack only via the accumulator. The two instructions for this are PHA and PLA.

PHA PusH Accumulator onto stack
PLA PuLl stack onto Accumulator

In Program 7.2, we use the stack as a temporary store whilst swapping the contents of &70 and &71.

```
10 REM SWAP VIA STACK
20
30 DIM Z%50
40 P% = Z%
50 [
60 OPT 1
70
80 .swap
90 LDA &70    \Put first number in A
100 PHA       \Save it on stack
110 LDA &71    \Put second number in A
120 STA &70    \Store it in first location
130 PLA       \Get back first number from stack
140 STA &71    \Store it in second location
150
160 RTS
170 ]
180
190 REM Test routine
200 ?&70 = &2D
210 ?&71 = &3E
220
230 PRINT "Numbers in &70 and &71 before are: " , ?&70, ?&71
240 CALL swap
250
260 PRINT "Numbers in &70 and &71 after are: " , ?&70, ?&71
```

Program 7.2

Test run

```
>RUN
Numbers in &70 and &71 before are:      45      62
Numbers in &70 and &71 after are:       62      45
```

The program has reversed the contents of the two locations using the stack as the temporary store that is needed in a swap operation.

7.4 Saving the registers on the stack

In general, when you call a subroutine, the registers will be in use and contain data which you will need again after the subroutine. But the subroutine itself needs to use the registers and so will destroy your data. When this is the case, you have to save the contents of the registers immediately before calling the subroutine. They can be saved on the stack.

The registers concerned are:

- The processor status register (flags)
- The accumulator
- The X register
- The Y register

Fortunately these are all 8-bit registers so they can be placed on to the stack in turn. We have already met the instruction to place the accumulator onto the top of the stack (PHA) but what about the other registers? There are two additional instructions available:

PHP Push Processor status onto stack
PLP Pull Processor status from stack

That still leaves the X and Y registers to transfer. These cannot be transferred directly. Instead, they have to be transferred one at a time into the accumulator and then pushed onto the stack with PHA i.e.

```
TXA
PHA
```

will put the X register onto the stack and

```
TYA
PHA
```

will put the Y register onto the stack.

SAQ 4

There are four registers to save. Does it matter in which order they are saved?

You don't always have to save all four registers since you may not have data in all of them. But unless you are sure that you don't need the data in them, save them all as follows:

PHP	Puts processor status onto stack
PHA	Puts A onto stack
TXA)Puts X onto stack
PHA)
TYA)Puts Y onto stack
PHA)

Then you can call a subroutine with JSR e.g.

```
JSR dosomething.
```

Once the subroutine has been executed, you must restore the saved registers to the state that they were in immediately before the subroutine call. This you do by reversing the above saving process.

SAQ 5

Write down the list of instructions to restore the registers to their state prior to the sub-routine call.

Program 7.3 shows the full saving and restoring process when all registers are in use.

```
...  
...  
...  
100 PHP  
110 PHA  
120 TXA  
130 PHA  
140 TYA  
150 PHA  
160 JSR dosomething ..... Go off to subroutine  
170 PLA  
180 TAY  
190 PLA  
200 TAX  
210 PLA  
220 PLP  
...  
...  
...
```

} Save registers

} Restore registers

Program 7.3

The state of the stack at line 160 (i.e. on calling the subroutine) will be as in Figure 7.8.

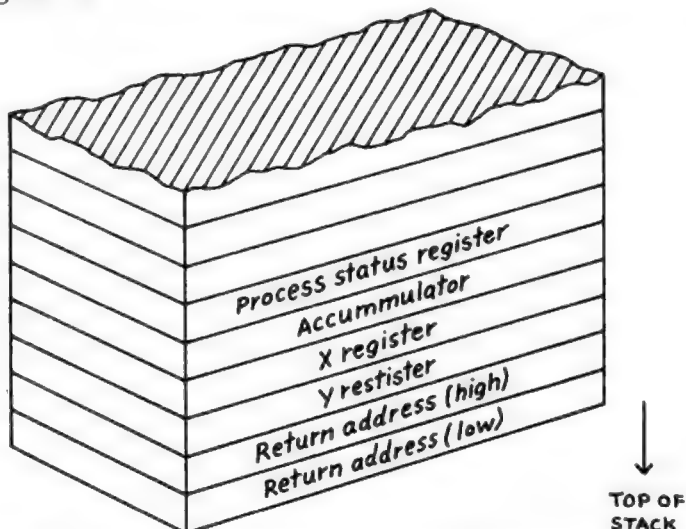


Figure 7.8 Stack when registers are saved before a subroutine

We have not bothered to save the registers before using the BBC Micro's built-in Operating System subroutines. This is because the built-in subroutines usually save and restore the relevant registers for you.

Built-in subroutines and the registers:

Key: 'Preserved' means the contents of the register before the call are still in the register after the call.

'Not relevant' means that you cannot use the state of such a flag to deduce anything about the result of the call. Nor can you assume that the flag has not been changed by the call.

Call	Action	State of registers on exit
OSWRCH {&FFEE}	Operating System Write Character Prints the character in A on screen (or printer)	A, X, Y preserved. C, N, V & Z not relevant. D = 0.
OSRDCH {&FFE0}	Operating System Read Character Takes a character from the keyboard and puts it into A.	A contains character. C = 0 means character was valid. C = 1 means invalid character. A, X, Y preserved. N, V, Z not relevant. D = 0.
OSNEWL {&FFE7}	Operating System New Line Produces a line feed and carriage return.	A, X, Y preserved. C, N, V, Z not relevant. D = 0.
OSASCI {&FFE3}	Operating System ASCII Prints the ASCII character of the character in A. If A = &0D then uses OSNEWL to produce a new line and carriage return.	A, X, Y preserved. C, N, V, Z relevant. D = 0.

('D' is the decimal mode flag which we haven't used in this book.)

SAQ 6

Program 3.7 printed 5 stars on the screen. The subroutine in Program 7.3 causes a delay of approximately 10 seconds. Incorporate it into Program 3.7 so that the stars are printed on the screen at 10 second

intervals. Make sure that you save the appropriate registers.

```
.delay
LDA #54
.waita
LDY #&FF
.waitb
LDX #&FF
.waitc
DEX
BNE waitc
DEY
BNE waitb
SEC
SBC #1
BNE waita
RTS
```

Program 7.4

7.5 Passing parameters via the stack

In many cases, a subroutine will perform some operation on data from an earlier part of the program. The subroutine must know where this data is to be found. One simple technique is to store the data in known memory locations. This is the method we have used so far in this course. Each time we wanted to pass parameters to a subroutine, we placed them in Zero Page. However, this method requires the subroutine to incorporate the addresses of those locations. Subroutines can be made more general by avoiding tying them too closely to the memory map of any particular computer. One method is to use the stack for data transfer. In this way the subroutine knows where to find the data without the programmer specifying particular memory locations.

There is one snag with this approach (you never get something for nothing!). Suppose we wanted to write Program 5.2 (the 8-bit division program) as a subroutine which, when called, finds the two 1-byte numbers on top of the stack. We would proceed as follows:

```
Put first number on stack
Put second number on stack
Call subroutine
...
...
```

But 'Call subroutine' will itself place the return address on the stack so that the top of the stack looks like Figure 7.9.

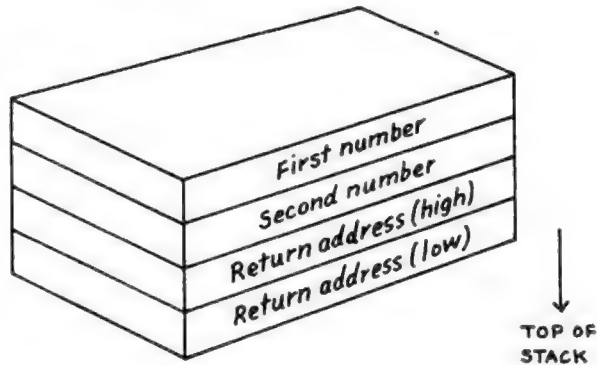


Figure 7.9 Parameters passed to a subroutine trapped under the return address

The parameters which we carefully placed on top of the stack ready for use by the subroutine are now buried by the return address from the program counter! One way out of this is to peel off the return address, rescue the parameters and then put back the return address. We do this as follows, using the X and Y registers as temporary stores for the peeled-off return address.

Step 1

PLA) Take return address (low) off stack
TAY) and save in Y
PLA) Take return address (high) off stack
TAX) and save in X

Step 2

...) Take parameters off the stack
...) – however many there are –
...) and put into suitable stores

Step 3

TXA) Put return address (high) into A
PHA) and push back onto stack
TYA) Put return address (low) into A
PHA) and push back onto stack

This process is illustrated in Figure 7.10.

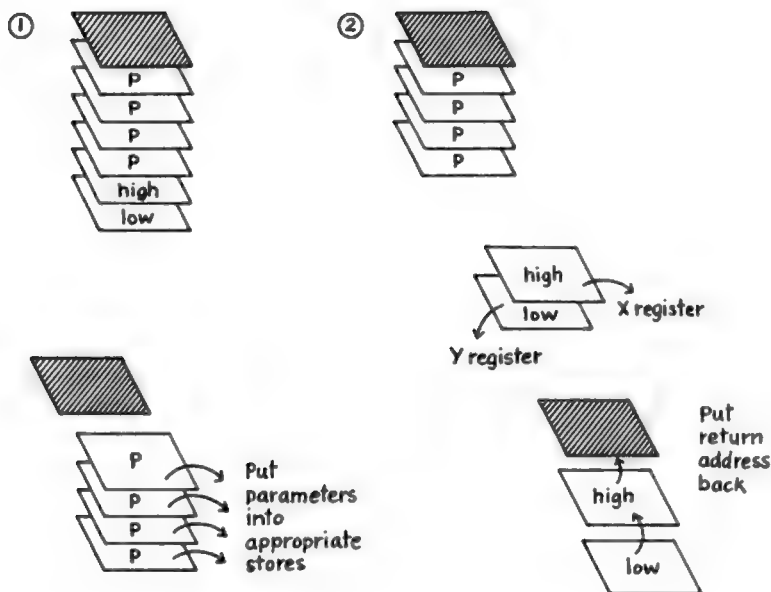


Figure 7.10 Saving the return address whilst taking parameters from stack

(Alternatively, the whole job can be done using the stack pointer but we have not covered that method in this book).

SAQ 7

Re-write Program 5.2 as a subroutine which expects to find the two 1-byte numbers on the stack.

Passing parameters via the stack is slow compared to other methods because, as you have seen, it involves a great many steps even to pass one parameter. It also carries the danger of an error in stack manipulation which will cause far more trouble than an error in addressing Zero Page. Consequently, this method is not recommended when other, quicker, machine-specific methods are available. Let's now look at some of the machine-specific methods in the BBC Micro.

7.6 Passing parameters via A%, X%, Y% and C%

When the BBC Micro enters an assembly language routine, the A, X, Y registers are set to the least significant bytes of A%, X% and Y% respectively. The carry flag is set to the least significant bit of C%. So, if you want to use this fact to pass parameters to a machine code subroutine, you must assign appropriate values to these variables in the BASIC part of the program immediately before entering the subroutine. (The A%, X%, Y% and C% you use must be BASIC's 'system' A%, X%, Y% and C% and not local variables created in a BASIC procedure or

defined function). Of course, if your program assigns its own values to the A, X, Y and C registers then you can ignore this feature (as we have done up to this point in this course).

SAQ 8

A program contains the lines

```
200 AX = &C0E5
210 X% = &56
220 Y% = 107
230 CX = &1F52
240 CALL prog
```

On entry to the assembly language program `prog`, what values are in:

- (a) the accumulator?
- (b) the X register?
- (c) the Y register?
- (d) the carry bit?

Program 7.5 demonstrates that the letters 'A', 'B' and 'C' have been transferred to the registers.

```
10 REM PARAMETER PASSING VIA AX, X% AND Y%
20
30 oswrch = &FFEE
40 DIM Z%50
50 P% = Z%
60 [
70 JSR oswrch
80 TXA
90 JSR oswrch
100 TYA
110 JSR oswrch
120
130 RTS
140 ]
150
160 REM Test routine
170 REM Parameters to be passed to assembly routine
180 AX = ASC "A"
190 X% = ASC "B"
200 Y% = ASC "C"
210
220 PRINT "Contents of A, X and Y registers: " ;
230 CALL Z%
240 PRINT
```

Program 7.5

Test run

```
>RUN
```

```
Contents of A, X and Y registers: ABC
```

This method of passing parameters can be used before CALL or USR i.e. it can be used before any entry to an assembly language routine.

SAQ 9

If passing parameters via A%, X%, Y% and C% is so straightforward, why is it not a complete answer to passing parameters?

7.7 Passing parameters via CALL

We used CALL to execute a machine code program but, in the BBC Micro, we can also use it to pass parameters to our program. This is a very powerful facility although quite complex to use. It is best explained by working step by step through an example.

To pass parameters via CALL, you follow the CALL program instruction with a list of the parameters to be passed, separated by commas e.g.

```
CALL program, S$, N%, A, H$(3)
```

Because the call is so complex, we are going to illustrate it by passing a single parameter. The program will take a BASIC string and print it out in reverse. Notice that we are not using a string at a fixed location (such as \$B) but an ordinary BASIC string, S\$, whose location in the memory is not fixed and not known to us. To pass this string to a machine code program, we proceed as follows:

Step 1

The string will have already been defined as a BASIC string, say S\$ = "WILL THIS STRING PRINT OUT BACKWARDS?"

Step 2

Pass the parameter to the assembly language program with the statement

```
CALL routine, S$
```

where 'routine' is the label for the assembly language program.

Step 3

The computer sets up a parameter block at location &600 upwards. In this block it places the information that you will need about the BASIC variables that you listed in your CALL statement. This information is as follows:

&600	Number of parameters received
&601) Address of first (low-byte)
&602) parameter (high-byte)
&603	Code for type of parameter received
&604) Address of second (low-byte)
&605) parameter (high-byte)
&606	Code for type of parameter received

etc. with 3 bytes of information for each parameter.

The codes are:

0	A single byte (e.g. ?N)
4	Four bytes (e.g. !N or N%)
5	Floating point variable (e.g. N)
&80	String at fixed location (e.g. \$S)
&81	Ordinary string (e.g. S\$)

If the parameter is a string, then the address in the parameter block is *not* the address of the string itself but of a 'string information block' which is yet another block of memory. A string information block is a 4-byte area of memory which contains:

	{ Address (low-byte) of string (high-byte)
	Bytes allocated to string
	Length of string

So that's what CALL does when followed by parameters. Now let's see how it behaves when it executes CALL routine, S\$ where S\$ = "Hello".

Step 1

S\$ = "Hello" (the variable must be defined before CALL)

Step 2

The computer will fill &600 - &603 with:

&600	1	(One parameter passed)
&601	Address (low-byte) of S\$ String Information Block	
&602	Address (high-byte) of S\$ String Information Block	
&603	&81	(Code for ordinary string)

Step 4

If we now go to the address given in &601 and &602, we will find a 4-byte block of information about S\$:

) Address of (low-byte)
) S\$ (high-byte)
	Number of bytes allocated*
5	Length of "Hello"

* When a string is first assigned to S\$, space is set aside for a string 8 bytes longer than the original string. Later in the program, a new string may be assigned to S\$ which may be of a different length to the first string assigned to S\$. The 'number of bytes allocated' will always be at least as great as the length of the current string allocated to S\$.

Program 7.6 shows this in action. The program prints out a string in reverse. By using CALL backwards, S\$ at line 500, we ensure that the relevant details of S\$ will appear in &600 to &603. (We labelled this

block par). We can then pick up the information that we need (lines 160 and 180). From there we go to the string information block relating to S\$ where we extract the second lot of information that we need including the true address of S\$. By storing this address in Zero Page, we are able to use indirect indexed addressing to get at each individual letter of S\$.

```

10 REM REVERSE A STRING
20
30 oswrch = &FFEE
40 osnewl = &FFE7
50 block = &70 : REM plus &71
60 string = &72 : REM plus &73
70 par = &600
80
90 DIM Z%50
100 FOR pass = 0 TO 3 STEP 3
110   P% = Z%
120   [
130     OPT pass
140
150     .backwards
160     LDA par+1      \Get low-byte of string information block
                        address
170     STA block      \Store it
180     LDA par+2      \Get high-byte of string information block
                        address
190     STA block+1    \Store it
200     LDY #0
210     LDA (block),Y  \Get low-byte of string address from block
220     STA string      \Store it
230     INY
240     LDA (block),Y  \Get high-byte of string address from block
250     STA string+1    \Store it
260     INY
270     INY
280     LDA (block),Y  \Get length of string
290     TAY            \Put it in Y
300     BEQ exit       \If string is length zero, exit
310
320     .loop
330     DEY            \Move pointer down string
340     LDA (string),Y \Get letter of string
350     JSR oswrch      \Output letter
360     CPY #0          \Finished?
370     BNE loop        \If not, loop back for next letter
380
390     .exit
400     JSR osnewl
410
420     RTS

```

```

430 ]
440 NEXT pass
450
460 REM Test routine
470 SS = "WILL THIS PRINT ALL RIGHT BACKWARDS?"
480 PRINT "SS is: " ' SS
490 PRINT ' "SS backwards is: "
500 CALL backwards, SS

```

Program 7.6

Test run

```

>RUN
SS is:
WILL THIS PRINT ALL RIGHT BACKWARDS?

SS backwards is:
?SDRAWKCB THGIR LLA TNIRP SIHT LLIW

```

SAQ 10

Write an assembly language program to print N% stars where N% is a parameter passed to the program by `CALL stars, N%`.

7.7 USR

USR offers a simple way of calling any assembly language routine which is required to return one value. However, that one value can carry several items of information.

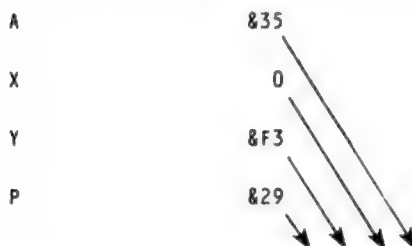
USR is slightly different from CALL. The statement

```
result = USR(prog)
```

will execute the assembly routine 'prog' and, at the end of the routine, assign a value to 'result' made up from the four separate values:

- The value in the processor status register.
- The value in the Y register.
- The value in the X register.
- The value in the accumulator.

If the values in the four registers at the end of 'prog' were:



then the result would equal: &29F30035.

This is demonstrated in Program 7.7 where the program puts 67, 12 and 199 into A, X and Y respectively.

```
10 REM USR DEMONSTRATION
20
30 DIM Z%50
40 PX = Z%
50 [OPT 1
60
70 .prog
80 LDA #67    \Put 67 into A
90 LDX #12    \Put 12 into X
100 LDY #199  \Put 199 into Y
110
120 RTS
130 ]
140
150 REM Test routine
160
170 PRINT "Output of use of 'USR' call: " ; "USR(prog)
```

Test run

```
>RUN
Output of use of 'USR' call: B1C70C43
```

Program 7.7

SAQ 11

Verify that the output is correct for the A, X and Y registers. What does it tell us about the processor status register?

Extracting information from USR

In the run of Program 7.7 we had to extract the information from the output visually. Usually, we want this information to be available to the program immediately after the USR call. There are two techniques. One is to re-write Program 7.7 as in Program 7.7a.

```
10 REM USR DEMONSTRATION 2
20
30 DIM Z%50, result 3
40 PX = Z%
50 [OPT 1
60
70 .prog
80 LDA #67
90 LDX #12
100 LDY #199
110
120 RTS
130 ]
140
150 REM Test routine
160
170 !result = USR(prog)
180 REM Now extract each byte from the memory block 'result'
190 PRINT "A = " ; result?0
200 PRINT "X = " ; result?1
210 PRINT "Y = " ; result?2
220 PRINT "P = " ; result?3
```

Program 7.7a

The other technique for extracting information from the result of USR is to use masking. Let's look at that next.

7.9 Masking

AND

AND is a logical operation between two binary numbers producing a result which contains 1's where both the operands contain 1's but leaving 0's in all other positions. This is best understood by imagining the two operands superimposed one on top of the other.

101 AND 110

Superimpose 110 onto 101:

101
110
!

Only position at which both numbers contain a 1.

Since bit 2 is the only bit position at which both numbers contain a 1, we find that:

101 AND 110 = 100

SAQ 12

What are:

- (a) 1010 AND 111 (b) 1010 AND 1111 (c) (any 8-bit number) AND &FF
(d) (any 8-bit number) AND &0

We can now use AND to extract information from the output of Program 7.7. We had

which represents

result = &B1 C7 0C 43
↑ ↑ ↑ ↑
P Y X A

To find A, we form

result AND &FF.

Since 'output' is a 32-bit number whilst &FF is an 8-bit number, 'AND &FF' 'masks off' all but the 8 least significant bits of &B1C70C43:

&FF = 000000FF
output = B1C70C43
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
Masked off by AND

So

A = output AND &FF i.e. &43.

To find Y we must mask off as follows:

B1C70C43
↑ ↑ ↑ ↑ ↑ ↑

This can be achieved by

output AND &FF00 = &0C00

and then dividing by &100.

i.e. X = (output AND &FF00) DIV &100 = &0C

It is important to use AND first, then DIV i.e.

(output AND &FF00) DIV &100 is correct

since

(output DIV &100) AND &FF00 is incorrect

The latter method produces incorrect extractions when the top bit of the number being extracted is set since DIV then treats it as a negative number.

To find Y we must mask off as follows:

B1C70C43

↑↑ ↑↑↑↑

which can be achieved with

Y = (output AND &FF0000) DIV &10000.

ORA and EOR

These are the remaining two logical operations.

ORA

ORA is the 'inclusive OR'. It is used as an operand between two numbers. If either number contains a '1' in a given bit position then that bit position will be 1 in the result.

e.g. 10001 ORA 1011 = 11011

Since bit 2 is the only bit position in which neither number has a 1.

and 10101 ORA 1011 = 11111 since between the two numbers, there is a 1 at every possible bit position.

In the 6502, of course, all numbers are 8-bit so the above results would be 8-bit in assembly language programs:

00010001 ORA 00001011 = 00011011

and

00010101 ORA 00001011 = 00011111.

ORA has one special use in assembly language programming and that is to ensure that one or more particular bits of an 8-bit number are set to 1. Suppose that we want bit 3 of 'number' to be a 1 and that the program does not require the specification of any of the other bit positions of number. We can achieve this by forming

&08 ORA number

since &08 = 00001000 and the result of any ORA operation with 00001000 must contain a 1 at bit 3.

SAQ 13

Evaluate the following:

(a) 101100 ORA 10000 (b) number ORA 11111111 (c) number ORA &0

EOR

EOR is the exclusive OR and behaves like the BASIC 'EOR' when used for bit masking. It is a logical operation between two numbers and the result contains a 1 at those bit positions where one and only one of the operands contains a 1. At all other bit positions the result contains a zero.

e.g.

10001 EOR 1011 = 11010 since at bits 1 and 3, the second number has 1's whilst the first does not. At bit 4, the first number has a 1 whilst the second does not. Elsewhere either both numbers have 1's or both numbers have 0's.

and

10101 EOR 1011 = 11110 since only at bit 0 do both numbers contain a 1. At all other bit positions there is a 1 in one of numbers but not in both.

As with ORA, the numbers will be 8-bit in the 6502 so the above results would be

00010001 EOR 00001011 = 00011010

and

00010101 EOR 00001011 = 00011110.

SAQ 14

What is number EOR &FF?

You can see from SAQ 14 that number EOR &FF will always form the complement of an 8-bit number. This is the main function of EOR in 6502 work.

AND, ORA and EOR in machine code

AND, ORA and EOR are used in assembly language programming with one of the two operands in A. The result of the operation is left in A. To perform any of the three logical operations between the numbers first and second, you write

```
LDA first
AND second
OR
```

```
LDA first
ORA second
```

```
OR
LDA first
EOR second
```

Hex/ASCII conversion

Quite often we want to enter hex numbers from the keyboard into a machine-code program. So far we have used ?& ... = & ... but that is inconvenient, to say the least. We can't enter the hex numbers at the

keyboard directly since neither OSRDCH nor OSWORD with A = 0 (described in Unit 8) will accept hex numbers. If we enter 'B3' then it is interpreted as the two characters 'B' and '3' and each will go into a separate memory location. If we wish to use OSRDCH or OSWORD with A = 0 to input hex numbers, we must convert them as they are input. Let's examine what's involved.

Suppose you have typed in 'B3' using OSRDCH and that '3' was stored in location 'low' and 'B' was stored in location 'high':

low	&33
high	&42

We want to get the computer to interpret these two bytes as the one byte &B3. It has to convert &33 into the four-bit binary number 0011 and &42 into the four-bit binary number 1011. Four bit-binary numbers are called 'nibbles'.

To convert ASCII codes for digits to binary, we need to convert as follows:

ASCII code	Hex number represented
&30	0
&31	1
&32	2
&33	3
&34	4
&35	5
&36	6
&37	7
&38	8
&39)Notice jump in	9
&41)sequence	A
&42	B
&43	C
&44	D
&45	E
&46	F

So to convert the Hex/ASCII numbers &42 and &33 to binary we subtract &30 from each number in turn and then an extra 7 from &42 because of the jump in the sequence (the jump arises from ASCII &3A to &40 being set aside for punctuation marks). The result is &B 3 which in binary is 1011 0011.

This conversion is summarised in Figure 7.11.

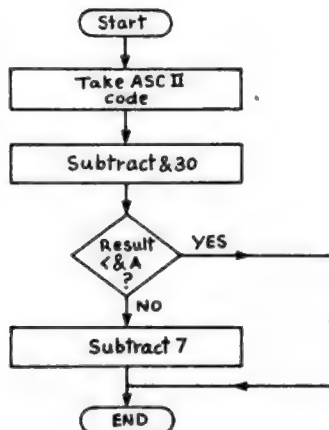


Figure 7.11 Hex/ASCII conversion

The program assumes that you have entered a hex number and that its low character is still in the accumulator. The high character is in the X register.

```

10 REM HEX/ASCII CONVERSION
20
30 lownibble = &70
40 DIM Z%50
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80   OPT pass
90
100  .hexasciiconversion
110  JSR convert \Convert low character
120  STA lownibble \Save it
130  TXA \Get high character
140  JSR convert \Convert it
150  ASL A \Shift it from
160  ASL A \low nibble
170  ASL A \position in A
180  ASL A \to high nibble position
190  ORA lownibble \ORA lownibble into A
200  RTS
210
220  .convert
230  SEC \Set carry for low-byte subtraction
240  SBC &30 \Take &30 off the character
250  CMP &A \Is result 10 or more?
260  BCC exit \If not, conversion is finished
  
```

```

270   SBC #7           \Else take a further 7 off nibble
280
290   .exit
300
310   RTS
320   ]
330   NEXT pass
340
350 REM Test routine
360 X% = &33
370 PRINT "Character put into X register: " CHR$(&33)
380 A% = &42
390 PRINT "Character put into A register: " CHR$(&42)
400 result = USR(hexasciiconversion)
410 A% = result AND &FF
420 PRINT "Contents of A register after conversion is " ; ~A%

```

Program 7.8

Test run

```

>RUN
Character put into X register: 3
Character put into A register: B

Contents of A register after conversion is 3B

```

Assignment G

1. Write an assembly language program that multiplies two 8-bit numbers, N% and M%, together where the numbers are passed to the program via CALL. Assume that the product will not exceed &FF.
2. Write an assembly language program to print LEFT\$(A\$,N%) so that it can be called and executed by CALL Left, A\$, N%.

Objectives of Unit 7

After studying this Unit, you should be able to:

- ☐ Use PHA and PLA to transfer data to and from the stack.
- ☐ Use PHP and PLP to save and restore the processor status register.

- ☐ Save the contents of the registers before entry to a subroutine.
- ☐ Restore the contents of the registers after the execution of a subroutine.
- ☐ Pass parameters to a subroutine via the stack.
- ☐ Pass parameters to a subroutine via A%, X%, Y% and C%.
- ☐ Pass parameters to a subroutine via CALL.
- ☐ Execute a subroutine using USR.
- ☐ Extract information from the value returned by USR.
- ☐ Apply AND, ORA and EOR to binary numbers.

Answers to SAQs

SAQ 1

The address to be saved is &200A (the address of the final statement in the program). This is saved as &20 (high-byte) followed by &0A (low-byte).

SAQ 2

At the start of &FFEE, the location &FFEE will be in the program counter so as to make the program jump to that location. But we don't know what numbers will be in the program counter during the execution of &FFEE. All we do know is that the program counter will be used by the subroutine at &FFEE as it will be by all subroutines.

SAQ 3

128. Remember that each call requires a value from the program counter to be placed onto the stack and that each program counter value is two bytes. If the stack was not in use at the start and not needed for any other purpose during the program, it could hold up to 128 return addresses.

SAQ 4

The order does matter for two reasons.

- (i) Since we use the A register to transfer the X and Y registers to the stack, we must save the A register before the X and Y registers.
- (ii) The operations TXA, TYA and PLA all affect the processor status register (e.g. they could set or clear the Z flag). Therefore, the processor

status register must be saved before the A, X and Y registers. So the order is:

- ☐ Save processor status register
 - ☐ Save accumulator
 - ☐ Save X register
 - ☐ Save Y register
- } In either
} order

SAQ 5

```
PLA
TAY
PLA
TAX
PLA
PLP
```

SAQ 6

```
10 REM LOOP
20
30 star = ASC "*"
40 oswrch = &FFEE
50 DIM Z%100
60 FOR pass = 0 TO 3 STEP 3
70   P% = Z%
80   [
90     OPT pass
100
110   .start
120   LDA #star           \Put '*' into accumulator
130   LDX #5             \Put 5 as counter into X
140
150   .print
160   JSR oswrch          \Print the '*'
170
180   \SAVE REGISTERS BEFORE DELAY SUBROUTINE
190   PHP                 \Save processor status onto stack
200   PHA                 \Save accumulator onto stack
210   TXA
220   PHA                 \Save X onto stack
230
240   JSR delay
250
260   \RESTORE REGISTERS AFTER DELAY SUBROUTINEA
270   PLA                 \Get X register content
280   TAX                 \Replace it
290   PLA                 \Get accumulator content
300   PLP                 \Get processor status content
```

```

310
320  DEX                \Decrease the counter by 1
330  BNE print         \Loop back if more '*'s to print
340
350  RTS
360
370  .delay
380  LDA #54
390  .waita
400  LDY #&FF
410  .waitb
420  LDX #&FF
430  .waitc
440  DEX
450  BNE waitc
460  DEY
470  BNE waitb
480  SEC
490  SBC #1
500  BNE waita
510  RTS
520  ]
530  NEXT pass
540
550 REM Test run
560
570 CALL start

```

Program 7.9

SAQ 7

```

10 REM DIVISION
20
30 DIM Z%100
40 dividend = &71
50 divisor = &70
60 remainder = &73
70
80 FOR pass = 0 TO 3 STEP 3
90  P% = Z%
100  [
110  OPT pass
120  LDA #5                \Divisor to be passed to subroutine
130  PHA                  \Push divisor onto stack
140  LDA #190              \Dividend to be passed to subroutine
150  PHA                  \Push dividend onto stack
160  JSR division
170  RTS
180

```

```

190 .division
200 PLA          \Pull low-byte of return address off
                stack
210 TAY          \Save in Y register
220 PLA          \Pull high-byte of return address off
                stack
230 TAX          \Save in X register
240 PLA          \Pull dividend off stack
250 STA dividend \Store at dividend
260 PLA          \Pull divisor off stack
270 STA divisor  \Store at divisor
280 TXA          \Transfer high-byte of return address
                to A
290 PHA          \Replace onto stack
300 TYA          \Transfer low-byte of return address
                to A
310 PHA          \Replace onto stack
320 LDA #0       \Clear A
330 LDX #8       \Shift counter
340 SEC
350
360 .shift
370 ASL dividend \Shift dividend left
380 ROL A        \Catch carry bit in A
390 CMP divisor  \Subtract divisor?
400 BCC next     \If no, then branch
410 SBC divisor  \Subtract divisor from A
420 INC dividend \Add one to quotient
430
440 .next
450 DEX          \Decrement shift counter
460 BNE shift    \Loop back if more shifts
470 STA remainder \Store remainder
480
490 RTS
500 J
510 NEXT pass
520
530 REM Test routine
540 CALL ZX
550 PRINT "Quotient = " ; ?&71
560 PRINT "Remainder = " ; ?&73

```

Program 7.10

Test run

```

>RUN
Quotient = 38
Remainder = 0

```

SAQ 8

- (a) The accumulator will contain &E5 i.e. the lsb of &C0E5.
- (b) The X register will contain &56. Since X% is a 1-byte number, the X register will contain X%.
- (c) The Y register will contain &6B. Since Y% is a 1-byte number, the Y register will contain Y%.
- (d) The carry bit will be set to 0. (The lsb of &1F52 must be 0 since &1F52 is an even number).

SAQ 9

First, we can, at most, pass 3 one-byte numbers and 1 one-bit number which will not be sufficient for some programs.

Second, the numbers have to be placed in A, X, Y and C and cannot be placed anywhere else by this method. In some programs, it might be necessary to re-locate some or all of the values as soon as they have been passed in order to put them where they are needed for a routine.

SAQ 10

```
10 REM CALL WITH PARAMETER PASSING
20
30 number = &70 : REM plus &71
40 par = &600
50 oswrch = &FFEE
70 star = ASC "*"
80 DIM ZX50
90
100 FOR pass = 0 TO 3 STEP 3
110   PX = ZX
120   [
130     OPT pass
140
150     .stars
160     LDA par+1          \Get low-byte of address of NX
170     STA number        \Store it in pointer (low-byte)
180     LDA par+2          \Get high-byte of address of NX
190     STA number+1      \Store it in pointer (high-byte)
200     LDY #0
210     LDA (number),Y    \Load NX into A
220     BEQ exit          \If NX = 0, skip star printing
230     TAY               \Copy it to Y
240     LDA #star         \Put a star into A
250
260     .next
270     JSR oswrch         \Output a star
280     DEY               \Decrement counter
290     BNE next          \Loop back if more stars to print
```

```

300 .exit
310
320 RTS
330 ]
340 NEXT pass
350
360 REM Test routine
370 REPEAT
380
390 INPUT "Enter number of stars (0 - 255): " N%
400 CALL stars, N%
410 PRINT
420
430 UNTIL FALSE

```

Program 7.11

Test run

```

>RUN
Enter number of stars (0 - 255): 20
*****
Enter number of stars (0 - 255): 5
*****

```

SAQ 11

output = &B1 C7 0C 43

&43 = contents of A
 &0C = contents of X
 &C7 = contents of Y
 &B1 = 10110001₂ = flags as below
 ↑↑↑↑↑↑↑↑
 NV-BDIZC

In this case, not much of the processor status information is of use, but we can see that the carry flag happened to be set. (The program itself neither set nor cleared it.) The Z flag is 0 as we would expect since the result of the last operation was not 0.

SAQ 12

(a) 1010 AND = 0010 = 10

111

↑

Position at which both numbers have a 1.

(b) $1010 \text{ AND } = 1010$

1111

↑ ↑

Positions at which both numbers have 1's.

(c) Any 8-bit number ANDed with &FF will be itself since &FF = 11111111.

(d) Any 8-bit number ANDed with 0 will be 0 since there are no 1's in 0.

SAQ 13

(a) $101100 \text{ OR } 10000 = 111100$

(b) number OR 11111111 = 11111111 = &FF

(c) number OR &0 = number (i.e. the operation has no effect)

SAQ 14

number EOR &FF = complement of number

e.g. $01101110 \text{ EOR } 11111111 = 10010001$ which is the complement of 01101110

UNIT 8

Operating System Calls

8.1 VDU calls

8.2 *FX calls

8.3 OSBYTE calls

8.4 Other operating system calls

8.1 VDU calls

Simple VDU calls

Many of the BBC Micro's VDU calls can be accessed via VDU in BASIC.

The calls vary in complexity (whether in assembly language programming or BASIC). The simplest are those that only require VDU followed by a number. These are:

VDU 2	Enable printer
VDU 3	Disable printer
VDU 4	Separate text/graphics cursors
VDU 5	Join text/graphics cursors
VDU 6	Enable VDU drivers
VDU 7	Make a short beep
VDU 8	Backspace cursor one character
VDU 9	Forwardspace cursor one character
VDU 10	Move cursor down one line
VDU 11	Move cursor up one line
VDU 12	Clear text area
VDU 13	Move cursor to start of line
VDU 14	Page mode on
VDU 15	Page mode off
VDU 16	Clear graphics area
VDU 20	Restore default logical colours
VDU 21	Disable VDU drivers
VDU 26	Restore default windows
VDU 30	Home text cursor to top left
VDU 127	Delete

(VDU drivers are commands which control output to the screen).

All of these can be implemented from an assembly language program by putting the VDU number into the accumulator and then calling JSR OSWRCH. Program 8.1 allows you to try these out one by one. Use only the numbers in the above VDU list.

```
10 REM ONE BYTE VDU CALLS
20
30 oswrch = &FFEE
40 DIM Z%25
50 P% = Z%
60 COPT 0
70 LDA &70      \Put VDU number into A
80 JSR oswrch \Execute VDU call
90 RTS
100 J
110
120 REPEAT
130   INPUT "Enter next VDU call number (99 to end). " N%
140   ?&70 = N%
150   IF N% <> 99 THEN CALL Z%
160   UNTIL N% = 99
```

Program 8.1

K Load Program 8.1.

We can't show the effects of this program on paper but you can try it for yourself. You will find that some calls produce no visible effect (i.e. VDUs 2, 3, 4, 5, 6, 16, 20 and 26). The effect of VDU 14 can only be seen when the cursor reaches the bottom of the screen, at which point the only way out is to press escape or shift. But if you have not reached the bottom of the screen then entering VDU 15 will cancel the effect of VDU 14. VDUs 6 and 21 are a similar pair. Once you have entered VDU 21, all VDU numbers except VDU 6 are ineffective.

More complex calls

All the other calls consist of a VDU number followed by some other numbers which give additional information to complete the VDU call. These numbers are the same in assembly language as in BASIC. You have probably used

```
MODE 5
```

and

```
COLOUR 2
```

The words **MODE** and **COLOUR** correspond to VDU numbers (22 = select screen mode and 17 = define text colour). Each has to be followed by a number which gives the particular Mode or colour to be selected. To use these calls in assembly language programs, the extra numbers have to go into the accumulator and be followed by JSR &FFEE. Here then is the routine for changing to Mode 5:

```
LDA #22          \VDU No for 'select screen mode'
JSR OSWRCH       \Execute VDU call
LDA #5           \No of mode required
JSR OSWRCH       \Output mode No.
```

and here is the routine for selecting colour 2:

```
LDA #17          \VDU No for 'define text colour'
JSR OSWRCH       \Execute VDU call
LDA #2           \No of colour required
JSR OSWRCH       \Output colour No.
```

(Changing Mode with VDU 22 does not change HIMEM. So VDU 22, 5 is not quite equivalent to BASIC's **MODE 5** command which does change HIMEM).

Neither of those look too bad but suppose that you wanted to change to Mode 5 and follow it with the selection of text colour 2. To do that you would need both routines - eight lines of programming! And that is for only moderately complex VDU calls. The better way makes use of the lists that you met in Unit 7.

Using lists

If you look at the two routines above for 'Mode 5, colour 2', essentially they consist of putting the numbers

22, 5, 17 and 2

into A individually and then doing a JSR OSWRCH after each one. So the program will be a great deal simpler if we put all the numbers into a list and then work through the list with a pointer. It will simplify the programming if (a) the first item in the list is the number of VDU bytes in the list and (b) the list of VDU bytes is arranged for reading from the end backwards. So, the list 22, 5, 17, 2 becomes 4, 2, 17, 5, 22, or, in hex,

&4, &2, &11, &5, &16

This is 'poked' into memory with

!S% = &05110204 and S%?4 = &16.

All we need now is a program that can work through this list and output each byte to JSR OSWRCH. This is shown in Program 8.2.

```

10 REM MULTIBYTE VDU CALLS
20
30 MODE7
40 DIM ZX50, SX8
50
60 oswrch = &FFEE
70 !S% = &05110204
80 S%?4 = &16
90 PX = ZX
100
110 [
120 LDX #0      \Pointer to start of VDU list
130 LDA S%,X    \Get first byte of list
140 TAX        \Put byte count in X register
150
160 .screen
170 LDA S%,X    \Get next VDU byte
180 JSR oswrch  \Output VDU byte
190 DEX        \Decrement byte counter
200 BNE screen  \Branch back if more bytes
210
220 RTS
230 ]
240
250 CALL screen

```

Program 8.2

[K] Load and run Program 8.2.

Program 8.2 may look a little long for what it does but on closer inspection it is very economical. The VDU calls are specified by S% = They are executed by lines 60 to 200 i.e. 5 lines. These 5 lines are all we need, however complex the series of calls becomes (unless we want to use more than 256 bytes!). As the series of calls gets longer, all we have to do is to change S% to make sure the DIM statement reserves enough space for the list. Here is a complex example using this method.

Example 1

Write a subroutine to select Mode 5 and to set up the screen to print green text on a red background.

Solution

We need a lot of bytes to do this, partly because green is not a default colour in Mode 5. We have to change one of the default colours to green. If we were solving this in BASIC we would write

```
MODE 5
VDU 19,2,2,0,0,0    Change colour 2 in mode 5 (yellow) to
                     logical colour 2 (green)
COLOUR 129           Background colour red
CLS                  Cover screen in red
COLOUR 2              Text colour
```

The equivalent VDU numbers and bytes are

BASIC	VDU Number to send	Bytes to send
MODE 5	&16	&05
VDU 19, 2, 2, 0, 0, 0	&13	&02, &02, &00, &00, &00
COLOUR 129	&11	&81
CLS	&0C	
COLOUR 2	&11	&02

Altogether that makes 13 bytes to be sent, so S% must allow for 13 bytes. Since we wish to use ! we need to make S% 16 bytes long.

```
S% ! 0 = &0C11020D
S% ! 4 = &00001181
S% ! 8 = &13020200
S% ! 12 = &00001605
```

The program to execute this series of calls is essentially the same as Program 8.2 and is shown as Program 8.3.

```
10 REM Demonstration of selecting mode 5
20 REM Putting yellow to green
30 REM Putting a red background on screen and green text
40 REM MULTIBYTE VDU CALLS
50
60 MODE7
70 DIM Z%50, S%16
80
90 oswrch = &FFEE
100 S%!0 = &0C11020D
110 S%!4 = &00001181
120 S%!8 = &13020200
130 S%!12 = &00001605
140 P% = Z%
150
160 [
```

```

170 LDX #0          \Pointer to start of VDU list
180 LDA S%,X        \Get first byte of list
190 TAX             \Put byte count in X register
200
210 .screen
220 LDA S%,X        \Get next VDU byte
230 JSR &FFEE      \Output VDU byte
240 DEX             \Decrement byte counter
250 BNE screen      \Branch back if more bytes
260
270 RTS
280 J
290
300 CALL screen

```

Program 8.3

As a final example, Program 8.4 shows tabbing using a VDU call and then prints a message at the tabbed position.

```

10 REM VDU CALL PLUS MESSAGE
20
30 DIM Z%50, S%4, W%30
40 oswrch = &FFEE
50 osnewl = &FFE7
60 $W% = "ALL DONE BY ASSEMBLER!"
70 S%!0 = &1F090D04
80 S%?4 = &0C
90 P% = Z%
100 [
110 LDX #0          \Pointer to start of VDU list
120 LDA S%,X        \Get number of bytes from list
130 TAX             \Transfer to X register
140
150 .tab
160 LDA S%,X        \Get next VDU byte
170 JSR oswrch      \Output VDU byte
180 DEX             \Decrement byte counter
190 BNE tab         \Branch back if more bytes
200
210 LDX #0          \Initialise letter counter
220
230 .word
240 LDA W%,X        \Get next letter
250 JSR oswrch      \Output to screen
260 INX             \Increment letter counter
270 CPX #LEN($W%)
280 BNE word        \Branch back if more letters
290 JSR osnewl      \New line & carriage return
300
310 RTS
320 J
330
340 CALL Z%

```

Program 8.4

 Load and run Program 8.4.

8.2 *FX calls

The BBC Operating System 1.0 has 54 *FX calls which control special effects of the Micro. They are listed on pages 418 – 419 of the *User Guide*. Most are followed by numbers which give the details of the call required e.g.

- *FX 5 Select printer type
- *FX 5, 1 Selects parallel printer output
- *FX 5, 2 Selects serial printer output
- *FX 8 Selects transmission rate to printer
- *FX 8, 1 75 baud
- *FX 8, 2 150 baud
- etc.

Let's see how these work in BASIC and then in assembly language.

Example 2

Select the serial printer output with 1200 baud transmission rate and switch the printer on. Print Hello and goodbye! and then switch the printer off.

Solution

First, in BASIC:

```
10 REM Select serial printer
20 *FX 5,2
30 REM Select 1200 baud transmission
40 *FX 8,4
50 REM Switch on printer
60 VDU 2
70 PRINT"Hello and goodbye!"
80 REM Switch printer off
90 VDU 3
```

Program 8.5

Now let's look at the assembly language solution.

*FX calls in assembly language

*FX calls in assembly language are done using OSBYTE called via &FFF4. OSBYTE does many different things, the exact call depending on the number in the accumulator. We shall be looking at its other uses in section 8.3 but first let's use it for *FX calls.

To execute *FX n1, n2, n3 you put n1 into the accumulator, n2 into the X register and n3 into the Y register. Then call OSBYTE. So *FX 5, 2 requires

```
LDA #5
LDX #2
JSR OSBYTE
```

(We don't have to bother about Y in this case since there are only two parameters to this call.)

That's all the new information that we need to use to write the printer program.

```
10 REM *FX CALLS
20
30 DIM S 20
40 DIM ZX50
50 $$ = "Hello and goodbye!"
60 oswrch = &FFEE
70 osbyte = &FFF4
80
90 FOR pass = 0 TO 3 STEP 3
100   PX = ZX
110   [OPT pass
120     LDA #5           \Put 5 into A for 5 of *FX 5, 2
130     LDX #2           \Put 2 into X for 2 of *FX 5, 2
140     JSR osbyte       \Execute *FX 5, 2 (RS432 printer)
150     LDA #8           \Put 8 into A for 8 of *FX 8, 4
160     LDX #4           \Put 4 into X for 4 of *FX 8, 4
170     JSR osbyte       \Execute *FX 8, 4 (1200 baud)
180     LDA #2           \Put 2 into A for VDU 2
190     JSR oswrch       \Execute VDU 2 (switch printer on)
200     LDX #0           \Initialise character counter
210
220     .sentence
230     LDA S,X          \Get a character of string $$
240     INX              \Increment character counter
250     JSR oswrch       \Output character
260     CMP #8D          \Was it 'return'?
270     BNE sentence     \If not, loop back for next character
280
290     LDA #3           \Put 3 into A for VDU 3
300     JSR oswrch       \Execute VDU 3 (switch printer off)
310
320     RTS
330   ]
340   NEXT pass
350
360 REM Test run
370
380 CALL ZX
```

Program 8.6

Test run

>RUN

Hello and goodbye!

Flushing the keyboard buffer

If the computer is not ready to accept keyboard input (e.g. it is busy doing some calculations), you can still type in at the keyboard. The characters that you type are stored in an area of memory called the keyboard buffer. At the next INPUT statement in the program, whatever is in the input buffer is accepted as input. If you want to prevent this, you precede the INPUT statement with *FX 21, 0 which flushes (empties) the input buffer. In BASIC you would write, say:

```
100 *FX 21, 0
110 INPUT "Enter next data " data$
```

(If your micro does not have operating system 1 – type *FX 0 to find out – then you will not be able to use *FX 21. As an alternative you can use *FX15 which flushes the currently selected input buffer.)

SAQ 1

Write an assembly language routine to flush the input buffer.

Other *FX calls

The following calls can be executed by the method given above. For full details of the parameters needed with each call, refer to the *User Guide*, pages 421 – 429.

*FX 0	Prints the number of the operating system in your micro.
*FX 1	-
*FX 2	Selects the input device (keyboard or RS432 port).
*FX 3	Selects the output stream.
*FX 4	Sets/re-sets the cursor editing keys.
*FX 5	Selects printer type.
*FX 6	Selects one ASCII code not to be sent to the printer.
*FX 7	Selects RS432 port receive baud rate.
*FX 8	Select transmission rate out of RS432 port.
*FX 9	Select flashing rate of flashing colours (first colour).
*FX 10	Select flashing rate of flashing colours (second colour).
*FX 11	Sets delay before keyboard auto-repeat starts.
*FX 12	Sets auto-repeat period.
*FX 13	Enable/disable various events.
*FX 14	Enable/disable various events.
*FX 15	Part of buffer flushing (see also *FX 21).
*FX 16	Select ADC channel(s).
*FX 17	Start ADC conversion.
*FX 18	Re-set user defined keys.
*FX 19	Wait until start of next TV frame.
*FX 20	Used in connection with re-definition of ASCII codes &20 to &FF.
*FX 21	Select buffer to flush.
*FX 124	Re-set escape flag.
*FX 125	Set escape key.
*FX 126	Acknowledge escape condition.

8.3 OSBYTE calls

The rest of the *FX calls are too complex to be executed in BASIC by the simple command *FX If they are used from BASIC, this has to be done with USR (see Unit 7). However, we are only going to deal with their use from assembly language programs. All OSBYTE calls (see the list in the *User Guide*, pages 429 – 441) are executed in exactly the same way as that described in section 8.2:

- Put the number of the call into the accumulator.
- Put any other parameters into the X and Y registers.
- JSR OSBYTE.

Many OSBYTE calls return information which is usually in the X and Y registers after the call has been executed. The general principles of OSBYTE calls can be explained by looking at a typical call: OSBYTE with A = &81 (Read a key with a time limit).

OSBYTE with A = &81

This is like BASIC's INKEY. It accepts a key-stroke provided that the key is pressed within a specified time limit. The period is set in centi-seconds by placing the lsb of the period in X and the msb of the period in Y e.g.

X register	Y register (max &7F)	Time period
0	0	0
&FF	0	2.55 secs
0	&10	40.96 secs
&FF	&7F	5 min 27.67 secs

The maximum value that may be placed in the Y register for this call is &7F.

After the call, the X and Y registers tell you what happened:

Y	Result
0	Key was pressed: character is in X.
&1B	Escape was pressed: must now 'acknowledge it' i.e. follow it by executing *FX 126
&FF	No key pressed.

Here is a program which waits 5 seconds for a key to be pressed. If a key is pressed, its character is displayed on the screen.


```

10 REM OSBYTE WITH A = &81
20
30 DIM ZX$50
40 osascii = &FFE3
50 osbyte = &FFF4
60 FOR pass = 0 TO 3 STEP 3
70   PX = ZX
80   [
90   OPT pass
100  LDA #21      \Put 21 into A for *FX 21
110  LDX #0       \Put 0 into X for *FX 21,0
120  JSR osbyte   \Execute *FX 21,0 (flush input buffer)
130  LDA #&81     \Put &81 into A for OSBYTE with A = &81
140  LDX #244     \Set delay to 244 centiseconds
150  LDY #1       \plus 256 centiseconds (total 5 seconds)
160  JSR osbyte   \Execute OSBYTE with A = &81 (wait for
                    key press)
170  TYA          \If Y = 0 then a key was pressed
180  BNE nocharacter \If no key press, check for escape
190  TXA          \Transfer character from X to A
200  JSR osascii  \Output character
210  RTS
220
230  .nocharacter
240  CPY #&1B     \Was escape pressed?
250  BNE exit     \If not, exit
260  LDA #126     \Otherwise acknowledge escape with *FX126
270  JSR osbyte   \Execute *FX 126
280
290  .exit
300  RTS
310  ]
320  NEXT pass
330
340  REM Test call
350
360 CALL ZX

```

Program 8.7

SAQ 2: OSBYTE with A = &87

This OSBYTE call reads the character at the current cursor position. On exit, X contains the character and Y the number of the current screen mode. Write a program to move the cursor to (x, y), read the character at that position and then to display it on the screen somewhere else. (Given the need to put values in X and Y at the start, it is best to call your routine with USR. Make the value that is returned by USR the ASCII code of the character which has been read).

OSBYTE with A = &8A (Put character into keyboard buffer)

This is a very useful routine since it provides a link between BASIC and assembly language. Let's first look at the call in BASIC.

***FX 138, 0, 65** places an 'A' (ASCII code 65) into the keyboard. We can use the call to place a series of characters into the keyboard buffer. Program 8.8 does this and then uses INPUT to see what is in the buffer. Not a useful thing to do but it's a way of seeing ***FX 138** at work.

```
10 REM Demonstration of *FX 138
20 REM Flush buffer at start
30
40 *FX 21,0
50 REM Put characters of 'Hello' into keyboard
60 *FX 138, 0, 72
70 *FX 138, 0, 69
80 *FX 138, 0, 76
90 *FX 138, 0, 76
100 *FX 138, 0, 79
110 REM End with return
120 *FX 138, 0, 13
130
140 REM Now see what is in the keyboard buffer
150 INPUT keyboard$
160 PRINT keyboard$
```

Program 8.8

***FX 138** offers a way of transferring data from BASIC to machine code and vice versa. You put the data from BASIC (say) into the keyboard buffer (remembering to flush the buffer first) and then take it out of the buffer with **OSRDCH**.

8.4 Other operating system calls

We have met some of these already (e.g. **OSWRCH**). A number of these calls are concerned with file handling (**OSFIND**, **OSGBPB**, **OSBPUT**, **OSBGET**, **OSARGS** and **OSFILE**) with which we are not dealing in this book. Let's look at the other calls.

OSRDCH (called via &FFE0)

This reads a character from the input stream. We have already used this to put characters from the keyboard into the accumulator. You can use **OSRDCH** to take characters from the RS432 stream by selecting that stream with ***FX 2, 1**.

OSASCI (called via &FFE3)

This routine writes the character in the accumulator to the current output stream (i.e. screen or printer). If the code in the accumulator is &D then the routine executes a 'new line and carriage return'.

OSNEWL (called via &FFE7)

Executes a 'new line and carriage return'.

OSWRCH (called via &FFEE)

Writes the character in the accumulator to the current output stream.

OSWORD (called via &FFF1)

OSWORD is actually twelve separate routines all called through &FFF1 with the routine selected depending on the value in the accumulator:

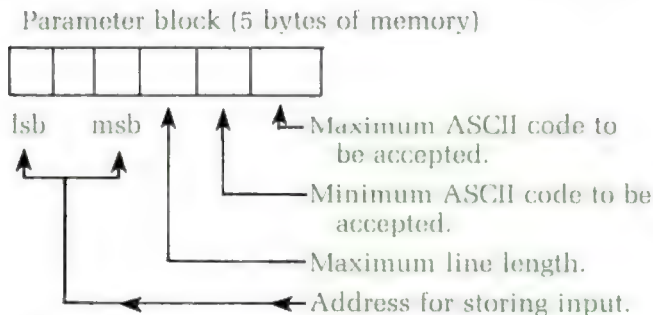
Value in A	OSWORD routine
0	Read a line of input from input stream.
1	Read the clock.
2	Set the clock.
3	Read the timer.
4	Set the timer.
5	Read byte of input/output processor memory.
6	Write a byte of input/output processor memory.
7	Sound.
8	Define envelope for sound.
9	Read pixel colour at a point.
A	Read dot pattern of a displayable character.
B	Read the palette value for a logical colour.

We shall look at OSWORD with A = 0, A = 1 and with A = 7.

OSWORD with A = 0

This does the same job as INPUT LINE in BASIC. It accepts all the characters typed in from the keyboard until 'return' is pressed and then stores them in a block of memory specified by you in the call. During this routine, you can use 'delete' and 'escape'.

The routine has to be given five parameters which is more than you can pass with the X and Y registers. Instead, the parameters are placed in a 'parameter block' designated by you. A parameter block is an area of memory set aside for passing information from one routine to another. For this routine, the block must contain the following details:



And, on top of all that information, we have to tell the routine where our parameter block is. We do this by putting the lsb of the address of the block into the X register and the msb into the Y register before calling the routine.

Example 3

Write a program to accept a line of input from the keyboard. The line

must be able to contain all printable ASCII characters, to be of maximum length 30 characters and to be stored in memory starting at &2200.

We shall use &70 to &74 as our parameter block.

Test run

>RUN

This was typed in with OSWORD

This was typed in with OSWORD

In a full working program you need to check the exit conditions of OSWORD with A = 0. They are:

Carry clear	means 'return' ended the input line
Carry set	means 'escape' ended the input line
Y	holds the length of the input line including any return

OSWORD with A = 1 (read clock)

This provides the same facility as TIME in BASIC. Since TIME soon becomes a large number, you need a 5-byte location to store its value. So, as with OSWORD with A = 0, you have to set aside a block of memory as a parameter block for the routine. You put the address of the block into the X register (lsb) and Y register (msb) and then execute the call with A = 1. Program 8.10 demonstrates its working with &70 to &74 as the parameter block into which TIME is deposited by the routine.

```
10 REM TIME
20
30 osword = &FFF1
40 DIM Z%50
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80     OPT pass
90
100    .time
110    LDX #&70          \Put lsb of parameter block address into X
120    LDY #0            \Put msb of parameter block address into Y
130    LDA #1            \Put 1 into A for OSWORD with A = 1
140    JSR osword         \Execute OSWORD with A = 1 to read time
150
160    RTS
170  ]
180  NEXT pass
190
200 REM Test routine
210 CALL time
220 PRINT "Time as read by routine: ";
230 time = !&70 + 256 * 4 * ?&74
240 PRINT time
```

Program 8.10

Test run

>RUN

Time as read by routine: 274156 .

OSWORD with A = 2 (set the clock)

In BASIC, you can set the clock with a statement such as `TIME = 0` or `TIME = 835138`. To do this in assembly language, you use **OSWORD** with `A = 2`. In this example, the 5-byte parameter block contains the value you wish to set `TIME` to. You put this value in the block, then put the address of the block into the `X` register (low byte) and `Y` register (high byte) and execute the call with `A = 2`.

SAQ 3

Write a program to set the clock to 0.

OSWORD with A = 7 (Sound)

This call produces the same effect as `SOUND` in BASIC. It needs 8 bytes of information so, again, you have to set up a parameter block but this time of 8 bytes:

Information	Bytes
Channel Number	lsb
Channel Number	msb
Amplitude	lsb
Amplitude	msb
Pitch	lsb
Pitch	msb
Duration	lsb
Duration	msb

Example 4

Write an assembly language program to play the note produced by the BASIC command `SOUND 1, -10, 53, 20`.

Solution

Using `&80` to `&87` as the parameter block, the block will need to contain:

Location	Value
<code>&80</code>	1
<code>&81</code>	0
<code>&82</code>	<code>&F6</code>
<code>&83</code>	<code>&FF</code>
<code>&84</code>	<code>&35</code>
<code>&85</code>	0
<code>&86</code>	<code>&14</code>
<code>&87</code>	0

Because the amplitude in the `SOUND` statement is negative (unless 0), we have to work out its two's complement (see Unit 1) in order to place it in the parameter block.

The program to play this note is Program 8.11. We have filled the parameter block using the indirection operator !. For more complex tunes, you would have to use a list-reading routine (see Unit 6) to fill the parameter block with the details for the successive notes.

```

10 REM SOUND
20
30 base = &80
40 REM Fill parameter block
50 base!0 = &FFF60001
60 base!4 = &00140035
70 osword = &FFF1
80 DIM Z%50
90 FOR pass = 0 TO 3 STEP 3
100 PX = Z%
110 [
120 OPT pass
130 LDX #&80      \lsb of parameter block address into X
140 LDY #0        \msb of parameter block address into Y
150 LDA #7        \7 into A for OSWORD with A = 7
160 JSR osword    \Execute OSWORD with A = 7 (sound)
170 RTS
180 ]
190 NEXT
200
210 CALL Z%

```

Program 8.11

Tunes

If we have to re-set all 8 parameters every time we change a note then the task is daunting. Usually, however, only one or two parameters change from one note to the next so we need only change those parameters. For example, if you look at the tune given on page 348 of the *User Guide*:

```

SOUND 1, -15, 97, 10
SOUND 1, -15, 105, 10
SOUND 1, -15, 89, 10
SOUND 1, -15, 41, 10
SOUND 1, -15, 69, 20

```

only the pitch of the notes changes, except for the last note, where pitch and duration change.

SAQ 4

Write a program to play the above tune.

Assignment H

1. Write an assembly language program to play a simple melodic phrase of your choice.

2. Write an assembly language program to clear the screen and display two messages at different points on the screen and in different colours. Make sure that the assembly language program sets the appropriate screen mode.

Objectives of Unit 8

After studying this unit, you should be able to write assembly language programs which:

- ☐ Execute VDU calls not requiring following bytes of information.
- ☐ Execute VDU calls with following bytes of information.
- ☐ Execute *FX calls with JSR OSBYTE.
- ☐ Execute more complex *FX calls which return information in the X and Y registers and in other parameter blocks.

Answers to SAQs

SAQ 1

```
10 REM Flushing the input buffer
20
30 osbyte = &FFF4
40 DIM Z%25
50 FOR pass = 0 TO 3 STEP 3
60   P% = Z%
70   [
80     OPT pass
90     .flush
100    LDA #21      \Put 21 into A for *FX 21
110    LDX #0       \Put 0 into X for *FX 21,0
120    JSR osbyte \Execute *FX 21,0
130    RTS
140   ]
150 NEXT pass
160
170 REM Test run
180 REM Pause to give you time to press a few keys
190 FOR I = 1 TO 5000 : NEXT
200 CALL flush
210 PRINT "Press return to see contents of input buffer after routine:"
220 INPUT T$
230 PRINT T$
```

Program 8.12

SAQ 2

```
10 REM CHARACTER READ
20
30 oswrch = &FFEE
40 osbyte = &FFF4
50 DIM Z%25
60 P% = Z%
70 [
80
90 .characterread
100 LDA #31          \Put 31 into A for VDU 31
110 JSR oswrch
120 TXA              \X coordinate for VDU 31
130 JSR oswrch
140 TYA              \Y coordinate for VDU 31
150 JSR oswrch        \That completes TAB(X, Y)
160 LDA #&87          \&87 into A for OSBYTE with A = &87
170 JSR osbyte        \Execute OSBYTE with A = &87
                        (read character on screen at X, Y)

180
190 RTS
200 ]
210
220 REM Test routine
230
240 REM Write a line of characters at line 2
250 CLS
260 FOR I = 0 TO 39
270   PRINT TAB(I, 2) CHR$(64 + RND(26)) ;
280 NEXT I
290
300 REM Read back the line using 'characterread'
310 FOR I = 0 TO 39
320   X% = I : Y% = 2
330   result = (USR(characterread) AND &FFFF) DIV &100
340   PRINT TAB(I, 22) CHR$(result) ;
350 NEXT
```

Program 8.13

SAQ 3

```
10 REM TIME SET
20
30 par1 = &70
40 par2 = &71
50 par3 = &72
60 par4 = &73
70 par5 = &74
80 osword = &FFF1
90 DIM z%50
100 FOR pass = 0 TO 3 STEP 3
110   P% = z%
120   [
130     OPT pass
140     .time
150     LDA #0           \Set parameter
160     STA par1         \block
170     STA par2         \to time
180     STA par3         \that you
190     STA par4         \wish to set clock to
200     STA par5         \ (zero in this case)
210     LDX #&70         \Put lsb of parameter block address
                        into X (will be destroyed by OSWORD call)
220     LDY #0           \Put msb of parameter block address
                        into Y (will be destroyed by OSWORD call)
230     LDA #2           \Put 1 into A for OSWORD with A=2
240     JSR osword       \Execute OSWORD with A = 2 to set time
250     RTS
260   ]
270   NEXT pass
280
290 REM Test routine
300 PRINT "TIME before using routine " ; TIME
310 CALL time
320 time = !&70 + 256 ↑ 4 * ?&74
330 REM Check that the clock has been re-set
340 PRINT "Current value of TIME is now " ; TIME
```

Program 8.14

Test run

```
>RUN
TIME before using routine 4033
Current value of TIME is now 2
```

(The 'current value' is not zero because of the time taken to exit from the routine).

SAQ 4

```

10 REM TUNE
20
30 base = &80
40 duration = &86
50 base!0 = &FFF10001
60 base!4 = &000A0061
70 osword = &FFF1
80 DIM ZX50
90 FOR pass = 0 TO 3 STEP 3
100   PX = ZX
110   [
120     OPT pass
130     LDA #&61           \Get next note change
140     JSR play           \Play note
150     LDA #105          \Get next note change
160     JSR play           \Play note
170     LDA #89           \Get next note change
180     JSR play           \Play note
190     LDA #41           \Get next note change
200     JSR play           \Play it
210     LDA #20           \Get next duration
220     STA duration      \Store it
230     LDA #69           \Get next note change
240     JSR play           \Play it
250     RTS
260
270     .play
280     STA &84            \Put new note in &84
290     LDX #&80           \Store lsb of parameter block address
300     LDY #0            \Store msb of parameter block address
310     LDA #7            \Put 7 into A for OSWORD with A = 7
320     JSR osword        \Execute OSWORD with A = 7 (sound)
330
340     RTS
350   ]
360   NEXT pass
370
380 REM Test routine
390 CALL ZX

```

Program 8.15

UNIT 9

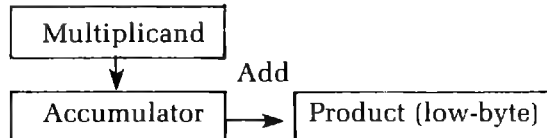
Tough stuff

- 9.1 16-bit multiplication
- 9.2 16-bit division
- 9.3 Sorting a BASIC array

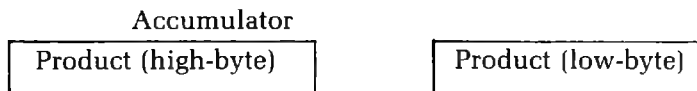
9.1 16-bit multiplication

Whilst 16-bit multiplication is more complicated than 8-bit, the principles are the same. What causes the trouble is the fact that the multiplicand will be 16 bits long and the product could be 32 bits long. This means that we need more memory locations and more movements of the numbers in memory.

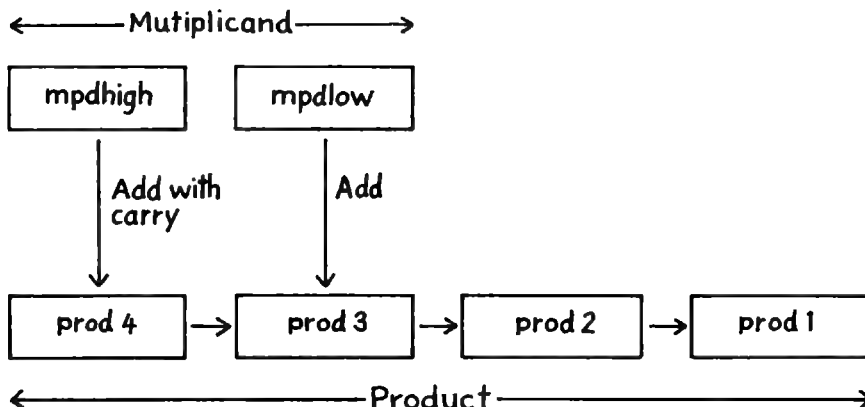
Before looking at the algorithm in detail, we shall compare the two methods schematically. Remember that 8-bit multiplication involved three registers:



We worked through each bit of the multiplier, starting with the lsb. If the bit was 1, we added the multiplicand to the accumulator and then rotated the lsb of the accumulator into the product's low-byte location. If the multiplier bit was a 0, only the shift took place. After 8 shifts the result was:



16-bit multiplication follows the same procedure but each 8-bit register/memory location is replaced by a 16-bit location:



The product consists of four bytes: prod1, prod2, prod3, prod4. To multiply, we work through the 16 bits of the multiplier. If a bit is a 1, we:

- Add mprlow to prod3
- Add mprhigh to prod4 with the carry of mprlow + prod3
- Shift prod4, prod3, prod2 and prod1 each one bit right.

If the multiplier bit is 0, we only perform the shifts. After 16 shifts, the 32-bit product is in prod4, prod3, prod2 and prod1 (highest byte to lowest byte).

The program

The memory allocation we shall use is as follows:

	High-byte	Low-byte
Multiplicand (mpd)	&71	&70
Multiplier (mpr)	&73	&72
Product (prod)	&77	&76, &75, &74

The program is given in Program 9.1. Compare this carefully with the 8-bit version in Program 5.1.

```

10 REM 16 BIT MULTIPLICATION
20
30 mpdlow = &70
40 mpdhigh = &71
50 mprlow = &72
60 mprhigh = &73
70 prod1 = &74
80 prod2 = &75
90 prod3 = &76
100 prod4 = &77
110
120 DIM Z%50
130 FOR pass = 0 TO 3 STEP 3
140   PX = Z%
150   [
160     OPT pass
170
180     .mult16
190     LDA #0
200     STA prod3      \Initialise high-byte of product to 0
210     STA prod4      \Initialise high-byte of product to 0
220     LDX #16        \Set bit counter
230
240     .next
250     LSR mprhigh     \Shift high-byte of multiplier
260     ROR mprlow      \Catch carry/ Next multiplier bit into
                       carry

```

```

270 BCC rotate      \Branch if multiplier bit was zero
280 LDA prod3       \Fetch 3rd byte of product
290 CLC             \Prepare to add
300 ADC mpdlow      \Add to low-byte of multiplicand
310 STA prod3       \Store new 3rd byte of product
320 LDA prod4       \Fetch 4th byte of product
330 ADC mpdhigh     \Add to high-byte of multiplicand
340
350 .rotate
360 ROR A           \Rotate 4th byte of product
370 STA prod4       \Store new 4th byte of product
380 ROR prod3       \Rotate 3rd byte of product
390 ROR prod2       \Rotate 2nd byte of product
400 ROR prod1       \Rotate 1st byte of product
410 DEX             \Decrement counter
420 BNE next        \Loop back if more bits to deal with
430
440 RTS
450 ]
460 NEXT pass
470
480 REM Test run
490 INPUT "Enter first number " first
500 ?&70 = first MOD 256
510 ?&71 = first DIV 256
520
530 INPUT "Enter second number " second
540 ?&72 = second MOD 256
550 ?&73 = second DIV 256
560
570 CALL mult16
580
590 PRINT "Result bytes (high to low) = " ?&74, ?&75, ?&76,
    ?&77`
600 PRINT "Result (decimal) = " ?&74 + ?&75 * 256 + ?&76 * 256
    ↑ 2 + ?&77 * 256 ↑ 3

```

Program 9.1

First test run with a multiplication which should give an 8-bit result: 12 * 13.

```

>RUN
Enter first number 12
Enter second number 13
Result bytes (high to low) =      156      0      0
Result (decimal) =      156

```

The result of 12 * 13 = 156 is correct.

Second test run which should yield a 32-bit product: 2038 * 8765.

```
>RUN
Enter first number 2038
Enter second number 8765
Result bytes (high to low) =      158      145      16
Result (decimal) = 17863070
```

Multiplying even more bits

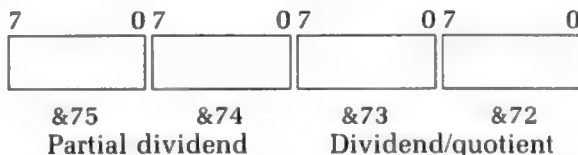
Going from 8 bits to 16 bits required two changes of method: first the need to keep all the bytes in memory; second, we had to watch the carry in the addition. But if we now want to move to 32-bit multiplication, no new principles are involved. We shall not deal with this in this book but you should find that what we have done will enable you to follow the listings of multi-bit multiplication programs in other sources.

9.2 16-bit division

The method we used for 8-bit division works with little modification for 16-bit division. The first modification concerns storage. Consider the arrangement we used for 8-bit division:



We can't directly use this for 16-bit division since the accumulator is only 8 bits long. We therefore have to use memory locations:



The second modification concerns the step of deciding whether or not we can subtract the divisor from the part of the dividend pushed into the partial dividend. With 8-bit division we used CMP but we can't use this on a 16-bit number. So we shall have to perform the subtraction and see whether the result is negative or not. If it is negative, then the partial dividend is smaller than the divisor and we forget it, proceeding to the next rotate. If the result of the subtraction is positive or 0 then the partial dividend is greater or equal to the divisor. So we accept the subtraction and increase the quotient by 1. The flowchart for this is shown in Figure 9.1.

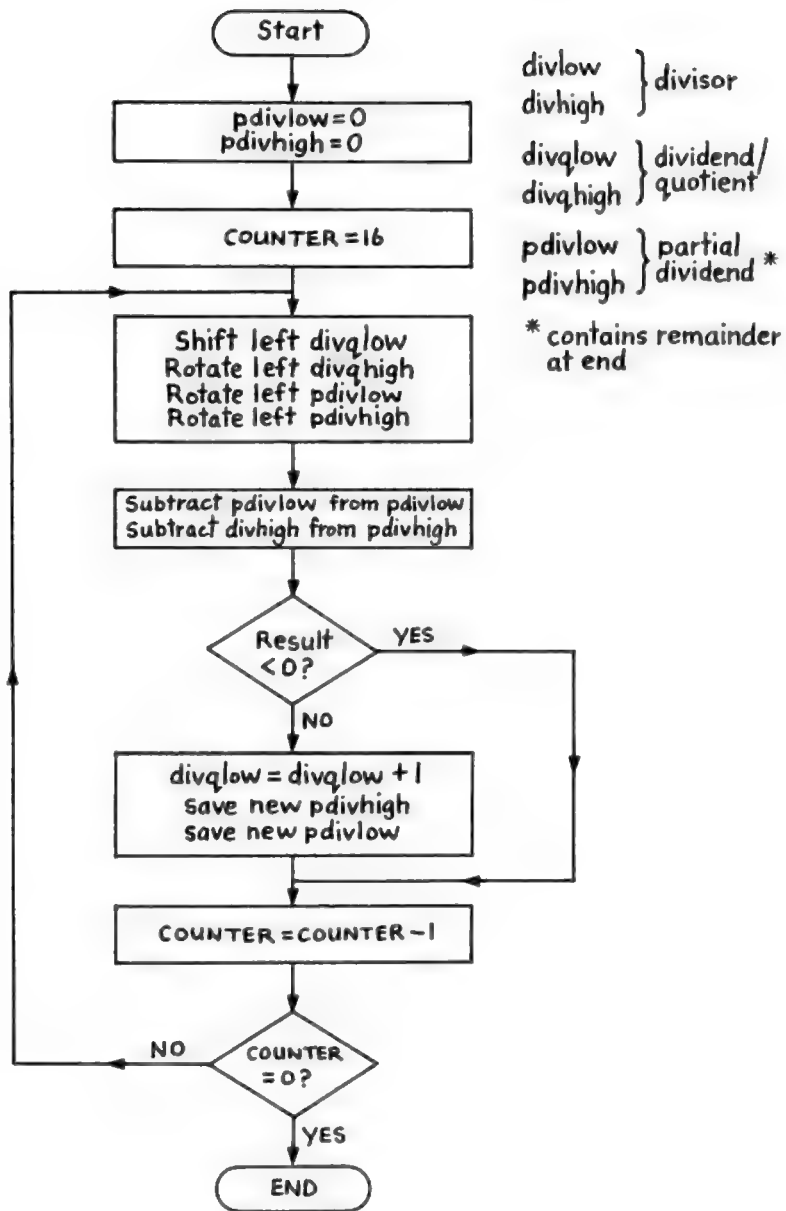


Figure 9.1 16-bit division

SAQ 1

Try writing the program for 16-bit division. Assume the divisor is in &70 (low-byte) and &71 (high-byte). Note: You need a place to store the 'low-byte partial dividend – low-byte divisor' subtraction result. You could use any memory location for this but the program will work fastest if you use the Y register.

9.3 Sorting a BASIC array

We have covered a lot in this book (but not everything!) – enough to now solve a recurrent practical problem of BASIC programming. We are going to write an assembly language program to sort a BASIC string array and so produce high speed sorting. We are still going to use the 'bubble' sort method introduced in Unit 6. There are much faster sorting methods, but they are harder to understand. However, even with the very inefficient 'bubble' method, our program is going to produce some very respectable sorting times. Here are the comparative results between a BASIC 'bubble' sort and the assembly language program that we are going to describe:

Items sorted	Typical time for sort	
	(seconds)	
	BASIC	6502
100 strings of length 5	26	0.84
100 strings of length 30	32	0.85
250 strings of length 30	201	5.32

The program is all about memory locations: where are the strings of the array? To understand what is happening, you have to know something about how the BBC Micro stores string arrays.

When the Micro executes the statement `DIM space$(10)`, it makes sure that it has enough room for 11 strings and, if it has, it sets up an information block to contain the details of the array. The actual strings of the array are spread all over the memory but the information block telling us (a) where the strings are, (b) how much memory has been set aside for them and (c) how long the current string is at any location, is in one continuous block of memory. The information block for a one-dimensional string array is shown in Figure 9.2. The block includes the name of the array less its first letter (`pace` instead of `space` in our example). The missing first letter is held in another part of the Micro's index to the BASIC variables.

You can see from this that, provided we can find the start address of the array, we can find all the addresses of all of the strings together with their lengths. You will remember that `CALL` can be used to place the addresses of BASIC variables into the parameter block at &600. So we can use `CALL` to find the address of the information block on the first

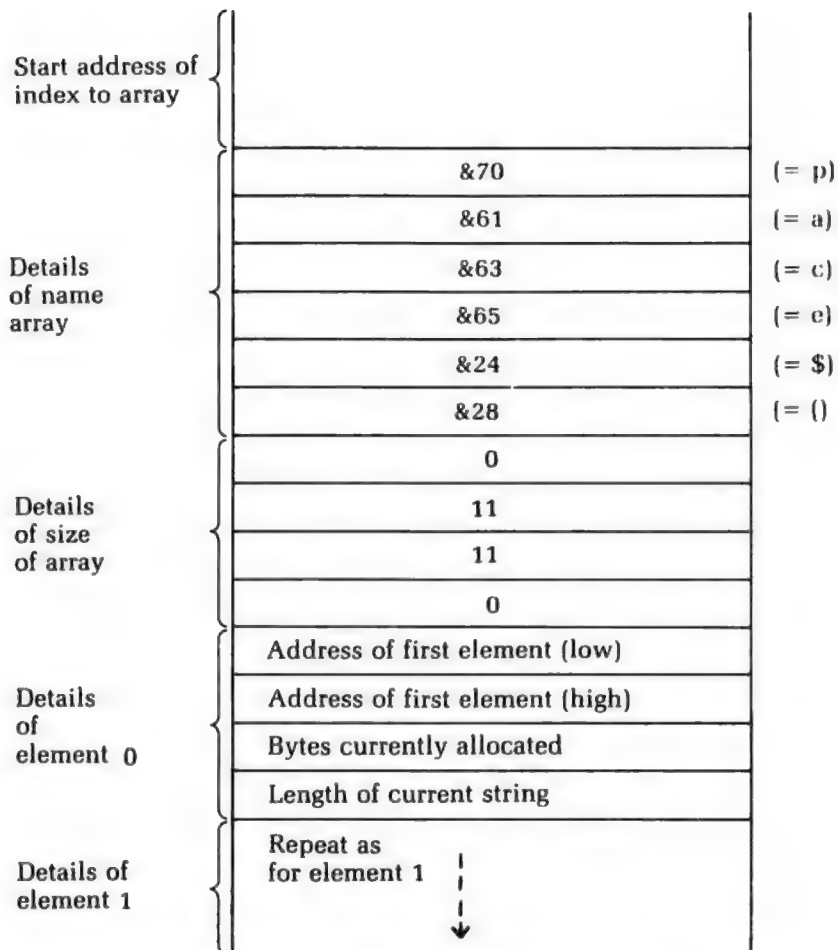


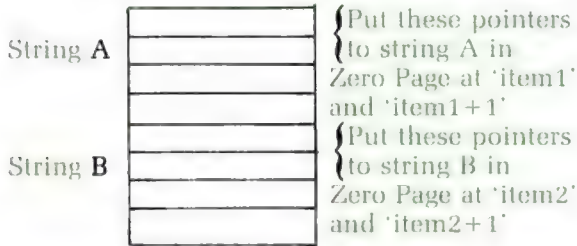
Figure 9.2 How the BBC Micro stores the details of an array

string in the array. We do this with `CALL program, A$(0)` where `A$()` is the array. (Don't forget that `A$(0)` is the first string and not `A$(1)`). `CALL program, A$(0)` will place the address of the string information block (SIB) on the element `A$(0)` into `&601` and `&602`. We don't need to use `CALL` for any further information on `A$(0)` since it is all there in the SIB. Similarly, we don't need to use `CALL` to get information on `A$(1)`, `A$(2)` etc. since their SIBs follow that of `A$(0)` in the memory.

From there on, the pattern is regular: each string in the array has a 4-byte String Information Block. We can therefore find the address of any string and so compare two strings to decide whether or not they are in the correct order. To do this, we are going to have to work through pairs of strings, letter by letter, starting at the first letter. The most

convenient way to do that is to put the addresses of the strings in Zero Page so that we can use indexed addressing to work through them:

STRING INFORMATION BLOCK



Indirect indexed addressing will enable us to take a character from string A and compare it with the same character in string B. If the character in string A is further on in the alphabet than the character in string B, we swap the two strings around and go on to the next pair. If the character in string A is the same letter as in string B, or before the string B letter in the alphabet, then we go on to look at the next letter along each string. This continues until either we find that we have to swap the two strings around, or we reach the end of the shorter of the two strings and find that they don't need to be swapped. If we have not found a swap to be needed by the time we have reached the end of the shorter string, then we need to swap the strings so that the shorter comes first e.g. of the two strings 'JAM' and 'JAMMY', 'JAM' should come first in an ordered list.

Swapping the strings involves a useful trick. The Micro only knows what, say, BS(3) is by the details contained in the String Information Block for BS(3). So, if we want to swap BS(3) with BS(4), we can do this by putting the information on BS(3) into the information block on BS(4) and vice versa. That involves swapping 4 bytes of information with another 4 bytes of information.

The program for the array sort is given in Program 9.2.

```

10 REM ARRAY SORT
20
30 MODE 3
40 HIMEM = HIMEM - 300
50 DIM space$(255)
60
70 baselow = &70 : REM Plus &71
80 item1 = &72 : REM Plus &73, &74, &75
90 item2 = &76 : REM Plus &77, &78, &79
100 elements = &7A
110 temp = &7B : REM Plus &7C
120 flag = &7D
130 shortest = &7E

```

```

140 par      = &600
150
160 FOR pass = 0 TO 3 STEP 3
170   PX = HIMEM
180   LOPT 0
190
200   .sort
210
220   \TAKE PARAMETERS FROM PARAMETER BLOCK
230   LDA par+4      \Get low-byte of address of 'elements%'
240   STA temp
250   LDA par+5      \Get high-byte of address of 'elements%'
260   STA temp+1
270   LDY #0
280   LDA (temp),Y   \Get value of 'elements%'
290   STA elements   \Store it in 'elements'
300
310   \MAIN LOOP OF SORT
320
330   .repeat
340   LDA #0
350   STA flag       \Clear swap flag
360   LDX elements   \Put number of elements into X
370   LDA par+1      \Get low byte of address of SIB of
                   string A
380   STA baselow    \Copy into another zero page location
390   LDA par+2      \Get high byte of address of SIB of
                   string A
400   STA baselow+1  \Copy into another zero page location
410
420   \PASSES THROUGH ON CURRENT BUBBLE
430
440   .nextpass
450   LDY #0
460   LDA (baselow),Y \Get low-byte of address of string A
470   STA item1      \Store it
480   INY
490   LDA (baselow),Y \Get high-byte of address of string A
500   STA item1+1    \Store it
510   INY
520   LDA (baselow),Y \Get number of bytes allocated to string A
530   STA item1+2    \Store it
540   INY
550   LDA (baselow),Y \Get length of string A
560   STA item1+3    \Store it
570   INY            \Increment pointer
580   LDA (baselow),Y \Get low-byte of address of pointer to
                   element B

```

```

590 STA item2      \Store it in zero page
600 INY            \Increment pointer
610 LDA (baselow),Y \Get high-byte of address of pointer to
                    element B
620 STA item2+1    \Store it in zero page
630 INY
640 LDA (baselow),Y \Get number of bytes allocated to
                    string B
650 STA item2+2    \Store it
660 INY
670 LDA (baselow),Y \Get length of string B
680 STA item2+3
690
700 .compare
710
720 \FIND LENGTH OF THE SHORTER OF THE TWO STRINGS
730 LDA item1+3    \Get length of string A
740 CMP item2+3    \Compare to length of string B
750 BCS second     \Branch if second is the shorter
760 STA shortest   \String A is the shorter
770 JMP checkdone
780 .second
790 LDA item2+3    \Get length of string B
800 STA shortest   \Store it in 'shortest'
810 .checkdone
820
830 \COMPARE THE TWO STRINGS, CHARACTER BY CHARACTER
840 LDY #0
850
860 .nextcompare
870 LDA (item1),Y  \Get a byte of string A
880 CMP (item2),Y  \Compare to same byte of string B
890 BCC swapdone   \If string B character > string A
                    character, skip swap
900 BNE swap       \If string A character > string B
                    character, then swap
910 INY            \Increment character pointer
920 CPY shortest   \Compare to length of shorter string
930 BEQ ordercheck \If end of shorter string, end character
                    comparisons
940 BNE nextcompare \Loop back to compare next character of
                    the two strings
950
960 \CHECK WHETHER SHORTER STRING IS FIRST OR NOT
970 .ordercheck
980 LDA item1+3    \Get length of string A
990 CMP item2+3    \Compare to length of string B
1000 BCC swapdone   \Skip swap if second string is

```

```

1010 BEQ swapdone      \not the shorter
1020
1030 \SWAP THE SIBs FOR THE TWO STRINGS
1040     .swap
1050     LDY #0          \Initialise pointer to characters
                        of the two strings
1060     LDA item2        \Get low-byte of address of string B
1070     STA (baselow),Y  \Store it in string A SIB
1080     INY
1090     LDA item2+1      \Get high-byte of address of string B
1100     STA (baselow),Y  \Store it in string A SIB
1110     INY
1120     LDA item2+2      \Get number of bytes allocated to
                        string B
1130     STA (baselow),Y  \Store it in string A SIB
1140     INY
1150     LDA item2+3      \Get length of string B
1160     STA (baselow),Y  \Store it in string A SIB
1170     INY
1180     LDA item1        \Get low-byte of address of
                        string A
1190     STA (baselow),Y  \Store it in string B SIB
1200     INY
1210     LDA item1+1      \Get high-byte of address of string A
1220     STA (baselow),Y  \Store it in string B SIB
1230     INY
1240     LDA item1+2      \Get number of bytes allocated to string A
1250     STA (baselow),Y  \Store it in string B SIB
1260     INY
1270     LDA item1+3      \Get length of string A
1280     STA (baselow),Y  \Store it in string B SIB
1290
1300     LDA #&FF
1310     STA flag          \Set swap flag
1320     \END OF CURRENT PASS
1330
1340     .swapdone
1350     CLC                \Prepare to move 'baselow' up 4 bytes
1360     LDA baselow        \Get 'baselow'
1370     ADC #4             \Add 4 bytes to it
1380     STA baselow        \Store new value
1390     BCC samepage
1400     INC baselow+1      \If there was a carry, add it to
                        basehigh
1410
1420     .samepage
1430     DEX                \Unsorted length is now 1 item less
1440     BEQ test

```

```

1450      JMP nextpass
1460
1470      .test
1480      LDA flag          \Did the last sweep through list
                           produce a swap?
1490      BEQ exit          \If 'no swap' then exit
1500      JMP repeat        \Otherwise, repeat the routine
1510
1520      .exit
1530      RTS                \Otherwise exit (array is sorted)
1540      ]
1550      NEXT pass
1560
1570      REM Test routine
1580      elements% = 255
1590
1600      REM Fill array with random letters
1610      FOR IX = 0 TO elements%
1620          length% = RND(5)
1630          FOR JX = 1 TO length%
1640              space$(IX) = space$(IX) + CHR$(64 + RND(26))
1650          NEXT
1660      NEXT
1670
1680      TIME = 0
1690
1700      CALL sort, space$(0), elements%
1710
1720      now = TIME
1730      PRINT "Sort time " ; now / 100 ; " seconds."
1740      FOR I = 0 TO elements%
1750          PRINT space$(I)
1760      NEXT
1770      END

```

Program 9.2

In Program 9.2, the sections of code headed `.nextpass` and `.swap` each consist of a repeated operation. In both cases it would be neater to shorten the code by turning them into loops. However, that would slow the program down which defeats the primary object of an array sort – speed.

Restrictions on the program

The maximum number of strings that can be sorted as the program stands is 255 since the X register holds the number of strings to be sorted. If you want to sort more, then you will need a 2-byte counter to replace X.

Assignment

There is no assignment for this unit.

Answers to SAQs

SAQ 1

```
10 REM 16 BIT DIVISION
20
30 divlow  = &70
40 divhigh = &71
50 pdivlow = &74
60 divqlow = &72
70 divqhigh = &73
80 pdivhigh = &75
90
100 DIM Z%50
110 FOR pass = 0 TO 3 STEP 3
120   PX = Z%
130   [
140     OPT pass
150
160     .div16
170     LDA #0
180     STA pdivlow      \Set low partial dividend
190     STA pdivhigh     \Set high partial dividend
200     LDX #&10         \Set bit counter
210
220     .next
230     ASL divqlow       \Shift dividend/quotient left
240     ROL divqhigh     \Shift dividend/quotient left
250     ROL pdivlow       \Shift bits into partial dividend
260     ROL pdivhigh     \Shift bits into partial dividend
270     LDA pdivlow       \Load low byte of partial dividend
280
290     SEC               \Prepare to subtract low bytes
300     SBC divlow        \Subtract low divisor
310     TAY               \Save result in Y
320     LDA pdivhigh     \Load high partial dividend
330     SBC divhigh       \Subtract high divisor
340     BCC done          \Skip subtraction if dividend is less than
                        divisor
350     INC divqlow       \Increment quotient
360     STA pdivhigh     \Save new high partial dividend
370     STY pdivlow       \Save new low partial dividend
380
390     .done
400     DEX
```

```

410   BNE next
420
430   RTS
440   ]
450   NEXT pass
460
470 REM Test run
480 INPUT "Enter dividend " first
490 ?&72 = first MOD &100
500 ?&73 = first DIV &100
510
520 INPUT "Enter divisor " second
530 ?&70 = second MOD &100
540 ?&71 = second DIV &100
550
560 CALL div16
570
580 PRINT "Quotient (low-byte/high-byte) " ; ?&72, ?&73
590 PRINT "Remainder (low-byte/high-byte) " ; ?&74, ?&75
600 PRINT "Quotient (decimal) = " ; ?&72 + ?&73 * &100
610 PRINT "Remainder (decimal) = " ; ?&74 + ?&75 * &100

```

Program 9.3

Test run

```

>RUN
Enter dividend 45643
Enter divisor 3462
Quotient (low-byte/high-byte) 13      0
Remainder (low-byte/high-byte) 125    2
Quotient (decimal) = 13
Remainder (decimal) = 637

```

UNIT 10

Round-up

10.1 Placing programs in memory

10.2 Placing data in memory

10.3 Programming errors

10.4 Program style

10.5 Saving machine code

10.1 Placing programs in memory

In Unit 2 we introduced the various methods of placing machine code programs into memory. Now that we have developed more programs we can expand on what we said there.

Method 1: space allocated by the BASIC program

Allocate the space with a statement such as:

```
DIM Z%50          (allocates 50 bytes labelled Z%)
DIM space 100      (allocates 100 bytes labelled space)
DIM prog%200       (allocates 200 bytes labelled prog%)
```

The space allocated in this way is just above TOP and the Micro ensures that BASIC's variables do not use this space. It is therefore an easy way of keeping an assembly language routine out of the way of any other memory usage.

Assemble the program in the reserved space by the statement `P% = ...` and execute it with `CALL Z%` or `CALL label`:

```
P% = Z% followed by CALL Z%
P% = space followed by CALL space
P% = prog% followed by CALL prog%
```

This method is ideal when one BASIC program is to have several assembly routines within it, each to be called at different points in the BASIC program. It leaves the assembler to take the decisions about memory allocation for the routines and you don't need to know where they are in order to call them. The routines are assembled into their memory blocks each time the BASIC program is run but they are saved as assembly language statements when the BASIC program is saved with the usual BASIC SAVE command.

Method 2: space chosen within BASIC's memory area

Allocate the space with a statement such as:

```
P% = &2000
P% = here (where here is a previously defined numeric variable)
```

Call the program by a statement such as:

```
CALL &2000
CALL here
```

This is a useful method if you want to keep several independent routines in memory at the same time. But you do have to be careful that they don't overlap, that their variables don't try to use the space of each other's programs and that the whole lot are clear of any BASIC program currently in memory and of its variables. All in all, a method full of

dangers when your machine code program is linked to a BASIC program!

Method 3: space chosen outside BASIC's reach

Find an area of memory that BASIC can't use or can be prevented from using for the time being. Here are some possibilities:

- ☐ Use &D00 – &DFF which is specially set aside for user routines (except on disk-based machines).
 - ☐ Use space which is reserved for something else which you know you are not going to need e.g. one of the buffers. (But you have to be certain that the buffer is not going to be used by the Micro for its intended purpose).
 - ☐ Move PAGE up and insert your routines between the old value of PAGE and the new value of PAGE.
 - ☐ Move HIMEM down and insert your routines between its new location and the default value for the Mode you are in. If you do this, you must remember not to change Mode since this would re-set HIMEM again.
- ☐ Place the routines in your chosen space by statements such as:

```
P% = &D00
P% = &E00    (assuming PAGE has been moved up in a system
              with only the Cassette Filing System).
```

```
P%=HIMEM    (assuming HIMEM has been moved down)
```

- ☐ Call the routines with statements such as:

```
CALL &D00
CALL &E00
CALL HIMEM
```

This method is ideal when you want a routine in memory that will stay there whilst several BASIC programs are used.

10.2 Placing data in memory

If your program includes data that needs to be saved with the program, then two main methods are open to you.

Method 1: leaving the data in the BASIC program

If your assembly routine is to be stored as assembly language then it will be stored as part of the BASIC program which assembles it. Any data needed by the assembly program can be stored within the BASIC program. It can be stored in the following ways:

- ☐ In indirection statements:

```
?&2000 = &67
```

```
&2000!0 = &8A54DF00
```

- ☐ In strings at fixed locations:

```
$S = "String to be saved with the program"
```

- ☐ In DATA statements which would then have to be read by the BASIC program before the data could be accessed by the machine code program.

Method 2: saving the data with the machine code

If you want to save the machine code rather than the BASIC assembly language program which created it, then any data needed by the program has to be saved with it. Since saving a machine code program involves saving a block of memory (see section 10.5), the data locations have to be consecutive with the program. It makes sense to either put all the data immediately before the program or immediately after.

Strings

There is a neat way of putting a series of items of string data into the area of memory immediately following the machine code. Each string is labelled `msg()` where the number of the string (or message) goes in the array bracket. The method also demonstrates how to include control codes in the strings so as to simplify their presentation on the screen. It uses `0` as an end-of-string delimiter instead of `&0D` as you have done so far.

```

10 REM Strings in assembly language
20
30 oswrch = &FFEE
40 DIM msg(3)
50 DIM ZX300
60 FOR pass = 0 TO 3 STEP 3
70   PX = ZX
80   [
90     OPT pass
100    LDY #0
110
120    .msg1loop
130    LDA msg(3),Y
140    BEQ msg1done
150    JSR oswrch
160    INY
170    BNE msg1loop
180    .msg1done
190
200    .exit
210    RTS
220    ]
230    PROCTEXT(1, "First message plus two line feeds" +
CHR$10 + CHR$10 + CHR$13)
240    PROCTEXT(2, CHR$31 + CHR$14 + CHR$12 + "Tab x,y then
second message")
250    PROCTEXT(3, CHR$10 + CHR$10 + CHR$13 + "Message three
with a beep" + CHR$7)
260    NEXT pass
270 CLS
280 CALL ZX
290 END
300
310 DEF PROCTEXT(N,A$)
320 msg(N) = PX
330 $msg(N) = A$
340 PX = PX + LEN(A$) + 1
350 PX? - 1 = 0
360 ENDPROC

```

Program 10.1

You can then print a particular message, say message 1, by a routine such as appears in Program 10.1 at lines 120 to 180.

Numeric data

This can be placed after the machine code program using a routine such as:

```
10 REM Numeric data storage
20
30 [
40
50
60 ]
70
80 REPEAT
90   READ N
100  IF N <> 999 THEN ?P% = N
110  UNTIL N = 999
120 END
130
140 REM Numeric data to be stored with machine code
150 DATA &12,0,&F3,&6,&8B,999
```

Program 10.2

10.3 Programming errors

Machine code programs do not produce error messages when they are running. The assembler itself produces some messages where errors can be picked up during assembly process. Attempting to place a number greater than &FF into an 8-bit location:

```
LDA #593
```

will produce the assembler error message 'byte'. Most of the time, however, assembly language programs either work or don't work with not much in the way of in-between results. The programs that don't work aren't very forthcoming about what is wrong. Checking through bugged programs requires careful work with pencil and paper to see what happens when particular assumptions are made about test data. When doing this you will be looking for results that you did not intend and for errors in the assembly language itself. Let's look at some of the commonest errors that you will need to watch out for.

Mixing numbers and addresses

e.g. writing

```
LDA &70 for LDA #&70
```

or

```
LDA #&70 for LDA &70
```

Both statements are correct so the assembler will not pick up an error. Only one, however, can be correct for the purpose you have in mind.

Mixing Hex and decimal

e.g. writing

LDX #56 for LDX #&56

or

LDX #&56 for LDX #56

Some of these errors can't be detected by the assembler. Where they can, you are likely to get an error message such as **Bad hex** or **Byte at ...**. This means that the number that you are trying to use as an operand is not in a form permitted by the assembler – usually because it is too large.

Confusions over address modes

It's easy to use the wrong mode for the purpose you have in mind. Here is a check list which should help you to choose the right mode for the job.

Operand required		Mode to use	Form
A number		IMMEDIATE	#number
An absolute address	16-bit	ABSOLUTE	&mmmm
	Zero page	ZERO PAGE	&mm
	16-bit indexed	ABSOLUTE IND X	&mmmm,X
		ABSOLUTE IND Y	&mmmm,Y
	Zero Page indexed	ZERO PAGE IND X	&mm,X
		ZERO PAGE IND Y	&mm,Y
An indirect address	Absolute	IND ABSOLUTE	(&mmmm)
	Indexed in Zero Page	INDEXED INDIRECT	(&mm,X)
	Indexed outside Zero Page	INDIRECT INDEXED	(&mm),Y
Address relative to present position		RELATIVE MODE	&mm
Accumulator		ACCUMULATOR	A
None required		IMPLIED	

Points to watch

- ☐ With all modes that state an address, check that the address will contain the required data.
- ☐ With Zero Page Indexed, make sure that the index is not too large. The index plus the base Zero Page address must not exceed &FF. (Indeed, with the BBC Micro where the free Zero Page locations are &70 to &8F, the base plus index must be in the range &70 to &8F).
- ☐ With indexed indirect and indirect indexed, the base address will be 16-bit but stored as two 8-bit numbers in Zero Page. These must be in adjacent locations (low-byte first) and you must remember to put the address of the lower location in the actual statement. You never need to address the higher of the two locations e.g.

Indirect indexed used to index from &3000:

Base address = &3000

Store in Zero Page: &70 = 0 (low-byte of &3000)

&71 = &30 (high-byte of &3000)

Address this as (&70),Y.

- ☐ With relative mode, don't try to branch more than 127 bytes forwards or backwards. Don't try to calculate the length of a branch – use labels and let the assembler do the calculation for you. If you try to branch too far, the assembler will tell you with an 'out of range' error message.

Error messages: if you are using indexed addressing and make an error that the assembler can detect in your index, you will get the message
`Index at`

Mistakes with the flags

Instructions which do not affect the flags

Remember that different instructions affect different flags and that some instructions do not affect any of the flags. It is all too easy to use a test after an instruction which does not affect the flag that you have chosen to test. For example, the increment instructions do not affect the C flag, so using BCC or BCS after INC, INX or INY will produce a result that you did not intend. (However the N and Z flags are affected.)

Similarly, the store instructions do not affect the flags at all so you must never attempt to branch on the basis of the result of a store instruction.

Incorrect use of the Z flag

The Z flag is 1 if the result of the last operation that could have affected

it was 0. The Z flag is 0 if the result of the last operation that could have affected it was not zero. It's easy to think that this is the opposite way round.

Incorrect use of the carry flag

With ADC

ADC is the only addition instruction in the 6502 instruction set, so all additions include a carry. If you do not want a carry, set the carry to 0 before the addition with the statement CLC (make C = 0).

With SBC

All subtractions include a borrow equal to the inverted value of the carry flag. So, if you do not want a borrow to take place, you must set the borrow to 1 before the subtraction. To do this you use SEC (makes C = 1).

With CMP, CPX and CPY

These three instructions affect the C flag but because 'compare' involves an internal subtraction, the C flag behaves as an inverted borrow. If the register (A, X or Y) is greater or equal to the value it is being compared to, the C flag is set to 1. C is only 0 if the register value was found to be less than the value to which it was being compared.

10.4 Program style

Assembly languages have the same deficiency as standard BASIC in that they are unstructured. The programmer is free to write messy and incomprehensible programs in such languages. BBC BASIC offers the opportunity to write structured programs using its many structured features, but no similar facilities are available in assembly language. If you wish to produce clear, understandable, assembly language programs, the style must come from you. Here are some guide-lines which may help you:

- First, plan your program on paper, writing out the whole program.
- Use only labels for memory locations in your operands. Never include an actual address in a program. (We have used actual memory locations on occasion but only because we were still building up the use of labels).
- All branches should be to labels. Never insert the number of bytes of jump required for a branch.
- Where a constant is to be used as an operand and that constant has some meaning, replace it by a meaningful label e.g. if &2A is to be used to print a star, then use the label `star = ASC "*" or eol = &0D`.
- Subdivide your program into sensible modules. Start all modules with a `rem` line i.e. a line that starts with `'` and has no other assembly statement on it. Put at least one empty line between modules.

- 11 Use plenty of clear comments. Labels and comments take up memory space during program development but are essential if you are to understand what you are doing. You can, of course, remove them from an actual working program if you need the memory space. Note that if you save the program as machine code, the comments and labels are lost automatically.
- 12 Stick to one assembly instruction per line.
- 13 Use empty lines to separate blocks of code (a line number followed by one press of the space bar will produce an empty line on the BBC Micro). Although there is no LISTO command in assembly language listings, you can synthesise it by indenting loops etc. with spaces.

10.5 Saving machine code

The programs in this course are BASIC assembly language programs and as such can be saved on cassette or disk in the same way as any other BASIC program. For example, to save Program 2.5 under the name SUB, you type

```
>SAVE "SUB"
```

and to load it back, you type

```
>LOAD "SUB"
```

It is also possible to save the machine code which has been generated by an assembly language program during the assembly process. If you look back at Program 2.5, its assembled version produced 8 bytes of machine code located between &2000 and &2007. In these 8 bytes are the numbers

```
A9 38 38 E5 70 85 71 60
```

They are a full machine code program and can be saved as such.

To save a machine code program, you use *SAVE. The syntax of the *SAVE command is

```
>*SAVE "PROG" ssss ffff eeee
```

where

ssss = the start address of the code to be saved (2000 for Program 2.5). Note that you must not put & in front of the hex number even though it is a hex number.

ffff = the end address of the code plus 1 (2008 for Program 2.5).

eeee = the execution address. This is optional and if you leave it out, the computer will assume that the execution address is the start address.

So, to save the machine code generated by Program 2.5, type

```
>*SAVE "SUB" 2000 2008
```

(If the program has appended data (e.g. text), assemble the program and then check the value of P% by typing `PRINT P%`. This will give you the end address of program plus data so that you can save both together.)

To load this back again, you can do one of two things:

Either: type `*RUN "SUB"`. This will cause the program to be loaded and run directly.

Or: type `*LOAD "SUB" 2000` which will cause the program to be loaded at &2000. It will not be executed until you type `CALL &2000`.

You will find that if you load a machine code program, you cannot use `LIST` to see it. The program is there, but invisible! The only way you can see it is by decoding it from its memory block with a special program called a disassembler. Such programs can be purchased for a few pounds.

BBC Micro character set*

Code Dec (Hx)	Char	Code Dec (Hx)	Char	Code Dec (Hx)	Char
0	(0) Does nothing	42 (2A)	*	84 (54)	T
1	(1) Next character to printer	43 (2B)	+	85 (55)	U
2	(2) Enable printer	44 (2C)	,	86 (56)	V
3	(3) Disable printer	45 (2D)	-	87 (57)	W
4	(4) Separate text/graphics cursors	46 (2E)	.	88 (58)	X
5	(5) Join text/graphics cursors	47 (2F)	/	89 (59)	Y
6	(6) Enable VDU drivers	48 (30)	0	90 (5A)	Z
7	(7) Make a short beep	49 (31)	1	91 (5B)	[
8	(8) Backspace cursor one character	50 (32)	2	92 (5C)	\
9	(9) Forwardspace cursor one character	51 (33)	3	93 (5D)]
10	(A) Move cursor down one line	52 (34)	4	94 (5E)	^
11	(B) Move cursor up one line	53 (35)	5	95 (5F)	_
12	(C) Clear text area	54 (36)	6	96 (60)	`
13	(D) Move cursor to start of line	55 (37)	7	97 (61)	a
14	(E) Page mode on	56 (38)	8	98 (62)	b
15	(F) Page mode off	57 (39)	9	99 (63)	c
16 (10)	Clear graphics area	58 (3A)	:	100 (64)	d
17 (11)	Define text colour	59 (3B)	;	101 (65)	e
18 (12)	Define graphics colour	60 (3C)	<	102 (66)	f
19 (13)	Define logical colour	61 (3D)	=	103 (67)	g
20 (14)	Restore default logical colours	62 (3E)	>	104 (68)	h
21 (15)	Disable VDU drivers	63 (3F)	?	105 (69)	i
22 (16)	Select screen mode	64 (40)	@	106 (6A)	j
23 (17)	Re-program display character	65 (41)	A	107 (6B)	k
24 (18)	Define graphics window	66 (42)	B	108 (6C)	l
25 (19)	Plot m, x, y	67 (43)	C	109 (6D)	m
26 (1A)	Restore default windows	68 (44)	D	110 (6E)	n
27 (1B)	Does nothing	69 (45)	E	111 (6F)	o
28 (1C)	Define text window	70 (46)	F	112 (70)	p
29 (1D)	Define graphics origin	71 (47)	G	113 (71)	q
30 (1E)	Home text cursor to top left	72 (48)	H	114 (72)	r
31 (1F)	Move text cursor to x, y	73 (49)	I	115 (73)	s
32 (20)	Space	74 (4A)	J	116 (74)	t
33 (21)	!	75 (4B)	K	117 (75)	u
34 (22)	"	76 (4C)	L	118 (76)	v
35 (23)	#	77 (4D)	M	119 (77)	w
36 (24)	\$	78 (4E)	N	120 (78)	x
37 (25)	%	79 (4F)	O	121 (79)	y
38 (26)	&	80 (50)	P	122 (7A)	z
39 (27)	'	81 (51)	Q	123 (7B)	{
40 (28)	(82 (52)	R	124 (7C)	
41 (29))	83 (53)	S	125 (7D)	}
				126 (7E)	~
				127 (7F)	Delete

* This is the same as the ASCII code for printable characters and the main control codes i.e. excluding the Mode 7 print controls above code 130.

6502 Instruction set: address modes, clock cycles and op codes

Code	Action	Flags affected NV BD I ZC	Acc	Imm'd	Zero page	Zero page x	Zero page Y	Abs	Abs	Abs	Y	Imp- lied	Rela- tive	(Ind. X)	(Ind.),Y	Ind- irect
ADC	Add operand to accumulator with carry	* * - - - - *	-	2(69)	3(65)	4(75)	-	4(6D)	4(7D)	4(79)	-	-	-	6(61)	5(71)	[1]
AND	AND operand with accumulator (result in A)	* - - - - - *	-	2(29)	3(25)	4(35)	-	4(2D)	4(3D)	4(39)	-	-	-	6(21)	5(31)	[1]
ASL	Shift operand left one bit	* - - - - - *	2(0A)	-	5(06)	6(16)	-	6(05)	7(1E)	-	-	-	-	-	-	-
BCC	Branch on carry clear (i.e. if C = 0)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(90)	-	-	[2]
BCS	Branch on carry set (i.e. if C = 1)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(80)	-	-	[2]
BEQ	Branch if last result was 0 (i.e. if Z = 1)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(F0)	-	-	[2]
BIT	Test accumulator with operand	[3] - - - - *	-	-	3(24)	-	-	4(2C)	-	-	-	-	-	-	-	-
BMI	Branch if result minus (i.e. if N = 1)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(30)	-	-	[2]
BNE	Branch if result not zero (i.e. if Z = 0)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(D0)	-	-	[2]
BPL	Branch if result plus (i.e. if N = 0)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(10)	-	-	[2]
BRK	Force break	- - - 1 - - -	-	-	-	-	-	-	-	-	-	7(00)	-	-	-	-
BVC	Branch on overflow clear (i.e. if V = 0)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(50)	-	-	[2]
BVS	Branch on overflow set (i.e. if V = 1)	- - - - - - -	-	-	-	-	-	-	-	-	-	-	2(70)	-	-	[2]
CLC	Clear carry (= set to 0) flag	- - - - - 0	-	-	-	-	-	-	-	-	-	2(18)	-	-	-	-
CLD	Clear (= set to 0) decimal mode flag	- - - - - 0	-	-	-	-	-	-	-	-	-	2(D8)	-	-	-	-
CLI	Clear (= set to 0) interrupt disable flag	- - - - - - -	-	-	-	-	-	-	-	-	-	2(58)	-	-	-	-
CLV	Clear (= set to 0) overflow flag	- 0 - - - - -	-	-	-	-	-	-	-	-	-	2(B8)	-	-	-	-
CMP	Compare operand with accumulator	* - - - - - *	-	2(C9)	3(C5)	4(D5)	-	4(CD)	4(DD)	4(D9)	-	-	-	6(C1)	5(D1)	[1]
CPX	Compare operand with X register	* - - - - - *	2(E0)	3(E4)	-	-	-	4(EC)	-	-	-	-	-	-	-	-
CPY	Compare operand with Y register	* - - - - - *	2(C0)	3(C4)	-	-	-	4(CC)	-	-	-	-	-	-	-	-
DEC	Decrement operand by 1	* - - - - - *	-	-	5(C6)	6(D6)	-	6(CE)	7(DE)	-	-	-	-	-	-	-
DEX	Decrement X register by 1	* - - - - - *	-	-	-	-	-	-	-	-	-	2(CA)	-	-	-	-
DEY	Decrement Y register by 1	* - - - - - *	-	-	-	-	-	-	-	-	-	2(88)	-	-	-	-
EOR	Exclusive OR operand with A (result in A)	* - - - - - *	-	2(49)	3(45)	4(55)	-	4(4D)	4(5D)	4(59)	-	-	-	6(41)	5(51)	[1]
INC	Increment operand by 1	* - - - - - *	-	-	5(E6)	6(F6)	-	6(EE)	7(FE)	-	-	-	-	-	-	-
INX	Increment X register by 1	* - - - - - *	-	-	-	-	-	-	-	-	-	2(E8)	-	-	-	-
INY	Increment Y register by 1	* - - - - - *	-	-	-	-	-	-	-	-	-	2(C8)	-	-	-	-
JMP	Jump to new address	- - - - - - -	-	-	-	-	-	3(4C)	-	-	-	-	-	-	-	5(6C)
JSR	Jump to subroutine	- - - - - - -	-	-	-	-	-	6(20)	-	-	-	-	-	-	-	-
LDA	Load accumulator with operand	* - - - - - *	-	2(A9)	3(A5)	4(B5)	-	4(AD)	4(BD)	4(B9)	-	-	-	6(A1)	5(B1)	[1]

Code	Action	Flags affected NV BD I Z C	Acc	Imm'd	Zero page	Zero x page	Zero Y	Abs	Abs	Imm'd	Ind	Ind	Ind	Ind
LDD	Load X register from operand	* - - - - *	-	2(A2)	3(A6)	-	-	4(AE)	-	4(BE)	-	-	-	[1]
LDY	Load Y register from operand	* - - - - *	-	2(A0)	3(A4)	4(B4)	-	4(AC)	4(BC)	-	-	-	-	[1]
LSR	Shift operand right one bit (Makes op bit 7 0)	0 - - - - *	2(4A)	-	5(46)	6(56)	-	6(4E)	7(5E)	-	-	-	-	-
NOP	Does nothing	- - - - -	-	-	-	-	-	-	-	2(EA)	-	-	-	-
ORA	Inclusive OR operand with A (result left in A)	* - - - - *	-	2(09)	3(05)	4(15)	-	4(0D)	4(1D)	4(19)	-	6(01)	5(11)	-
PHA	Push accumulator onto stack	- - - - -	-	-	-	-	-	-	-	-	-	-	-	-
PHP	Push processor status onto stack	- - - - -	-	-	-	-	-	-	-	-	-	-	-	-
PLA	Pull accumulator from stack	* - - - - *	-	-	-	-	-	-	-	-	-	-	-	-
PLP	Pull processor status from stack	* * * * *	-	-	-	-	-	-	-	-	-	-	-	-
ROL	Rotate operand one bit left	* - - - - *	2(2A)	-	5(26)	6(36)	-	6(2E)	7(3E)	-	-	-	-	-
ROR	Rotate operand one bit right	* - - - - *	2(6A)	-	5(66)	6(76)	-	6(6E)	7(7E)	-	-	-	-	-
RTI	Return from interrupt	* * * * *	-	-	-	-	-	-	-	6(40)	-	-	-	-
RTS	Return from subroutine	- - - - -	-	-	-	-	-	-	-	6(60)	-	-	-	-
SBC	Subtract operand from accumulator with carry	* * - - - *	2(E9)	3(E5)	4(F5)	-	-	4(ED)	4(FD)	4(F9)	-	6(E1)	5(F1)	[1]
SEC	Set (= set to 1) carry flag	- - - - - 1	-	-	-	-	-	-	-	-	-	-	-	-
SED	Set (= set to 1) decimal mode flag	- - - - - 1	-	-	-	-	-	-	-	-	-	-	-	-
SEI	Set (= set to 1) interrupt disable flag	- - - - - 1	-	-	-	-	-	-	-	-	-	-	-	-
STA	Store accumulator	- - - - -	-	-	2(85)	4(95)	-	4(8D)	5(9D)	5(99)	-	6(8D)	6(91)	-
STX	Store X register	- - - - -	-	-	2(86)	-	4(96)	4(8E)	-	-	-	-	-	-
STY	Store Y register	- - - - -	-	-	2(84)	4(94)	-	4(8C)	-	-	-	-	-	-
TAX	Transfer accumulator into X register	* - - - - *	-	-	-	-	-	-	-	2(AA)	-	-	-	-
TAY	Transfer accumulator into Y register	* - - - - *	-	-	-	-	-	-	-	2(A8)	-	-	-	-
TSX	Transfer stack pointer into X register	* - - - - *	-	-	-	-	-	-	-	2(BA)	-	-	-	-
TXA	Transfer X register into accumulator	* - - - - *	-	-	-	-	-	-	-	2(8A)	-	-	-	-
TXS	Transfer X into stack pointer	- - - - -	-	-	-	-	-	-	-	2(9A)	-	-	-	-
TYA	Transfer Y register into accumulator	* - - - - *	-	-	-	-	-	-	-	2(98)	-	-	-	-

Entry format: e.g. 4(8D) = 4 clock cycles (op code, 8D)

Key:

- * Not used in this book.
- Flag affected; could become 0 or 1.
- 0 Flag not affected.
- 0 Flag becomes 0.
- 1 Flag becomes 1.

Notes:

- [1] Add 1 to clock cycles if operation crosses a page boundary.
- [2] Add 1 to clock cycles if branch is to same page.
- Add 2 to clock cycles if branch is to another page.
- [3] V becomes bit 6 of operand; N becomes bit 7 of operand.

Imm'd
Page
Abs
Imp
Rel
Ind

Timing and cycles

The speed of operation of a microprocessor is controlled by an electronic clock. This clock sends out pulses which determine the precise moment when events can take place. As a result, it is possible to be very precise about how long any particular 6502 instruction will take to execute. The BBC Micro uses the 6502A microprocessor which has a 2 Megahertz clock i.e. a clock that sends out 2 000 000 pulses each second. One 'cycle' of the clock therefore takes 0.5 microseconds.

In the 6502 instruction set table in Appendix 2, you can see how many cycles each instruction takes e.g.

LDA in immediate mode takes 2 cycles

LDA in indexed indirect mode takes 6 cycles

(Some timings depend on exactly what happens during an instruction, as explained in two of the footnotes to the instruction set table).

If you look at the Zero Page column you will see that, for any given instruction, the Zero Page timings are 1 cycle shorter than the absolute mode column timings i.e. it is always quicker to address Zero Page than a 16-bit address.

These timings can be used for two purposes:

(a) to design programs that take a precise period of time e.g for timers, delays and to synchronise one computer with another.

(b) to compare one method of programming a routine with another in order to determine which is fastest.

Here are two examples of how you can use the timings to calculate the time a program will take to run.

Example 1

The program for adding two two-byte numbers was given in Program 2.7:

```
LDA &70
CLC
ADC &72
STA &74
LDA &71
ADC &73
STA &75
RTS
```

We can now write down how many cycles each instruction will take:

LDA &70	3	(Zero page mode)
CLC	2	
ADC &72	3	(Zero page mode)
STA &74	3	(Zero page mode)
LDA &71	3	(Zero page mode)
ADC &73	3	(Zero page mode)
STA &75	3	(Zero page mode)
RTS	6	
TOTAL	26 cycles	

So, addition of two-byte numbers takes 13 microseconds provided we use Zero Page. Or, put another way, we can do 76,923 two-byte additions in 1 second.

Example 2

Program 4.2 involved a branch. Because we have a loop, some instructions are executed more than once. We have therefore to consider (a) how long each instruction takes and (b) how many times each instruction is executed. One of the instructions (BNE) takes 3 or 4 cycles depending on whether or not the branch is to the same page. Unless your timings are very critical, it's safe to assume that the branch will be to the same page. So the program and its timings are:

	Cycles	Times executed	Totals
LDY #0	2	1	8
STA (&80),Y	6	1	
INY	2	255	
STA (&80),Y	6	255	
CPY #0	2	255	3315
BNE next	3	255	
RTS	6	1	6
		TOTAL	3329

This shows that one page of memory can be cleared in 1664.5 microseconds, a rate of 600 pages per second.

Further reading

COLL, J. *The BBC Microcomputer User Guide* ed. by David Allen, BBC, 1982.

The most useful sections for this course are:

39 Indirection operators; 40 HIMEM, LOMEM, TOP and PAGE;
42 *FX calls and OSBYTE calls; 43 Assembly language; Appendix:
Memory maps; Appendix: Memory map assignments

LEVENTHAL, L.A. and SAVILLE, W. *6502 Assembly Language Sub-routines* Osborne/McGraw-Hill, 1982.

Contains a useful reference section on 6502 programming techniques.
But the bulk of the book is 40 standard routines which can be used in
your programs.

RUSTON, J. *The BBC Micro Revealed Interface*, 1982.

Contains some material on how the BBC Micro stores its variables. At
the time of its writing, however, Operating System 0.1 was the only
available Operating System.

ZAKS, R. *Programming the 6502* 3rd rev.ed. Sybex, U.S. 1981.

Chapter 4 which summarises the 6502 instruction set (100 pages of it!)
is the best reference work to have at your side when writing programs.

ZAKS, R. *6502 Applications Book* Sybex, U.S. 1981.

ZAKS, R. *6502 Games* Sybex, U.S. 1981.

BBC
£7.25

Beyond Basic

Beyond Basic has been written to provide an easy introduction to assembly language on the British Broadcasting Corporation Microcomputer. The book incorporates a number of special features to help you get started with assembly language programming:

- ☐ The book is built around practical programming problems – often taking tasks you can do in BASIC and asking 'How can we do that in assembly language?'
- ☐ The 6502 instructions are introduced as they are needed to solve problems.
- ☐ Each step is illustrated by fully worked examples.
- ☐ As you work through the book, you are given plenty of opportunity to check your progress with self-assessment questions. Complete answers are given to all these questions.
- ☐ A cassette tape of all the main programs is also available.

The book will teach you how to use most of the 6502 instruction set and how to write programs to: perform arithmetic calculations, use loops and decisions, create lists and tables and perform complex operations on BASIC variables (e.g. sorting a BASIC array). The book also deals in detail with how to access the British Broadcasting Corporation Microcomputer operating system and includes assembly language routines for graphics and sound.

